DISK ENCLOSED

Get Professional-looking Menus With MenuScript!

AC's TECH / AMIGA

For The Commodore

Volume 2 Number 2
US $14.95 Canada $19.95

AMIGA VOICE RECOGNITION

- Programming the Amiga in Assembly Language, Part 2
- Dynamically Allocated Arrays
- Implementing an ARexx Interface in Your C/Program, Part 2
- Copper Programming
- Blit Your Lines
- Animated Busy Pointer
- Iterated Function Systems for Amiga Computer Graphics
- Keyboard I/O from Amiga Windows

02
0 74470 78640 5

WE DELIVER!

That's right, we deliver! We deliver the latest news on new products and happenings in the Amiga market and development community. We bring you the hottest products in accurate up-to-date reviews. We supply the tips, tricks and helpful hints on how to get the most from your Amiga. Let Amazing Computing bring you the best of the Amiga. Call now for fast delivery!

Call
1-800-345-3360
to have Amazing Computing delivered to you!

Contents

Volume 2, Number 2 AC's TECH/AMIGA

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
One Current Place
Fall River, MA 02722

AC's TECH AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Senior Copy Editor:	Paul Larrivee
Copy Editor:	Timothy Duarte
Video Consultant:	Frank McMahon
Art Director:	Richard Hess
Photographer:	Paul Michael
Illustrator:	Brian Fox
Editorial Assistant:	Torrey Adams
Production Assistant:	Valerie Gamble

ADVERTISING SALES

Advertising Manager: Wayne Arruda

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

SPECIAL THANKS TO:
Richard Ward & RESCO

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Current Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for $44.95; in Canada & Mexico surface, $52.95; foreign surface for $56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

Startup-Sequence

Changing Times

AC's TECH for the Commodore Amiga was the first disk-based technical magazine geared toward high-end Amiga users and Amiga developers. It is now the only magazine of its kind available for the Amiga. Recently, our only competitor in the technical market, *Amiga World's Tech Journal*, announced that they would be ceasing production. You may think this is good news for *AC's TECH* and for the most part, it is. But *AW's Tech Journal* was a good publication and helped to bring much needed support to the growing number of Amiga programmers and developers. Like *AC's TECH*, it brought valuable insights into the inner workings of the Amiga.

Filling the void

AC's TECH will fill the gap created by the loss of *AW's Tech Journal*. We have stepped up efforts to gather much needed information from developers and from Commodore. We are also searching for members of the Amiga Community who wish to share their knowledge of the Amiga with others interested in the technical aspects of Amiga computing. We are going to work even harder to remain the best publication ever created for the Amiga Technical Community.

Over the course of the next few issues, you will see an impressive change in *AC's TECH*. We will feature more articles than ever before. TECH will provide the latest and best information available to make your dream projects come true. It will feature a special section providing information on product development from start to finish. Moreover, it will show you how to bring your creations to market and how to make a profit from your Amiga. The new TECH will provide views from throughout the Amiga Technical Community. We won't let you down.

Whether it's *AC's TECH*, *Amazing Computing*, or *AC's GUIDE*, you know can be sure that there is a hard working team behind those magazines looking out for your best interests and providing you with all the information you need to make your Amiga complete.

Changing of the Guard

Also with this issue, we say good-bye to a good friend and co-worker, Ernest Viverios, Jr. Ernie has taken leave of the company to pursue other interests in the computer industry. Ernie was part of the founding staff of *Amazing Computing for the Commodore Amiga*. He has served as Associate Editor of that magazine as well as Editor of *AC's TECH* and *AC's GUIDE*. We will continue Ernie's tradition of excellence that helped to make *AC's TECH* and its sister publications number one. We wish Ernie the best of luck in his future endeavors.

Information Exchange

Knowledge is the key to success. *AC's TECH* is dedicated to instilling you with the knowledge you need to achieve optimal use of your Amiga. If you have created a program you feel would be useful to others, know some tricks and short-cuts to programming, are a master at a particular language, or have an idea for an interesting hardware project, call us. Likewise, if you are an Amiga Developer and would like to share your experience and knowledge with the rest of the Amiga Community, let us know! We are always looking for great article ideas, so share the wealth of Amiga knowledge and write for *AC's TECH*.

Jeffrey Gamble
Editor

PROGRAMMING THE AMIGA IN ASSEMBLY LANGUAGE

MACROMANIA

In the first article of this series we did a lot of extra writing when typing the programs. Most of this was repeating several key phrases over and over. A simple way to cut down on all of this repetition is to use a Macro—sort of a glorified GOSUB routine. The two differences between macros and sub-routines are:

1. A macro is repeated each time you call for it in the program. This creates a larger program but lets the program run more quickly. I think it also makes the source code more understandable.
2. Values can be passed to a macro very easily; you can even write your macro to assemble only part of itself depending on the value passed to it. You may pass numbers, phrases, labels, etc. to a macro, all with the same procedure. The key point to remember is that a macro only affects the writing of a program, not its operation. Once it's in the program, it won't change.

The first line of a macro contains the title of the macro on the left followed by a space or tab and MACRO. The routine the macro assembles follows, typed as a regular assembly program. Any branches and labels within the macro are written as NAME\@. Finally, all macros must end with a tab and ENDM (END Macro). Pass values to the macro within the program by typing a tab, including the macro name followed by a tab and any variables to be passed, each separated by a comma. These variables may be numbers, addresses, routine names, library routines, etc.; just separate each with a ",". The first variable is referred to within the macro by calling it "\1", the second is "\2", etc.

It's a matter of choice whether you want to use the library offset names or their actual values. I feel the program is more readable when using library names, but that does take some extra typing. Just try to be consistent. Macros can be very habit-forming. You can make them as complex as you want, and one macro may even call another. I wouldn't try to be too "cute" with them. Even if no one else will be reading them, you will look back at a program written a few months earlier and wonder what you meant. Always include the values expected as a remark alongside the macro name for reference. In fact, I keep a separate notebook of all of my macros to include the name, what variables are expected, what registers it uses, what's required in the program, and any other pertinent information. Even if it's just a one-time macro, I still record it; you never know when you may need it again—there's no sense in re-inventing the wheel.

Macros may be saved in a macro file to be used with any program. But since macros are so specific, it makes sense to save them in different files depending on with which major library they are used. In this way only the macros you need are called for at the beginning of the program with tab, INCLUDE MACRONAME.I. Since most routines need the EXEC Library, I made this my general macro file, and the majority of programs will include it.

EXEC MACROS

Let's take a look at Listing 1, EXEC MACROS. Notice that most of the EXEC Library offsets are included first. You could have all of the offsets for all of the libraries in one file and "include" it, but I like to have just the ones I need available and it's one file fewer. Since memory types are important to several EXEC routines, I also define the type of memory and equate them to their numerical values. Just to refresh your memory:

Public memory—any available type (fast first, chip second) Chip memory—for screen displays or sound

Fast memory—for arrays and variables;

Clear—sets allocated memory to zero

The first actual macro is called SYSLIB. It automatically puts the location of the EXEC Library address in register a6. Since this macro is used in conjunction with the next two macros, let's look at OPENLIB. To open the DOS Library, we need only type OPENLIB DOS,DONE. This will store the library name in register a1, the lowest acceptable version (0) in register d0, and call the macro SYSLIB, passing the value OpenLibrary. The SYSLIB macro will then execute that routine returning the library location in register d0. OPENLIB will store that value in "dosbase"; the "\1" is actually converted to read "dos" so "\1base" becomes "dosbase". If the library wasn't found, a zero would be returned and the macro would branch to the address in "\2" or, in this case, DONE. Already we've replaced seven lines of commands with just one line! The next four macros are necessary since the current version of A68K does not support a BLO (Branch if LOwer) or BHS (Branch if Higher or Same) command; the macros allow you to use these commands replacing then with the equivalent BCS (Branch if Carry Set) or BCC (Branch if

PART II—READIN', WRITIN' & 'RITHMETIC

by William F. Nee

Carry Clear). The MEMORY macro will tell you how much of chip or fast memory is available. Finally, EVENPC will create a blank word value, forcing addresses to start at a location divisible by four. There are other macros within this file but I'll wait until we actually use them before I discuss them. Copy this file from the magazine disk to the ASSEMBLER disk I discussed in Part I of this series as EXECMACROS.I.

DOS MACROS

Next let's review the DOS MACROS, Listing 2. DOSLIB will execute whatever DOS routine you pass to it. STYLE is our first example of conditional assembly. IFNC checks to see if two strings are not the same; IFC checks to see if they are the same. So the first line of the macro will look at the first value passed and if it is not zero, it will print in that style. If the value had been a blank followed by a comma (,2,3 for example) the '\1' would have been a blank and that part of the macro would not assemble. In the same manner the macro also checks for a foreground and background color. You could pass from one to three values as long as you have the commas separating them (NORMAL,,3 for example). Notice the single quotes around '\1' convert it to a string.

PRINT is a macro that prints a message just as I discussed previously in Part I. You can either pass the message length as the second variable or omit it and let the macro compute the length. Since there can be only up to two variables passed, you don't need the comma if you just pass the message name. The macro PRINTHEXR will print the contents of any register in HEX format (Base 16) just as I discussed in Part I. The only change I made is to save registers a0, and d0-d3 with MOVEM.L (MOVE Memory), pushing them on to the stack; they are recalled in the same way at the end of the macro. PRINTDECR prints the contents of any register as an unsigned decimal number. A 10-byte buffer is reserved at the end of the file and the macro stores ASCII values of numbers there from right to left. The number in d1 is shifted left; the value in d0 is rotated left and the left-bit from d1 that went to the C and X bit in the status register becomes the first bit in d0. Using the ROXL.L command means to include that X bit as a replacement bit for the one lost during the rotate. Any time d0 exceeds a value of 9, that value is decreased by 10. Since we're shifting d1, we can also use it as an exponent counter and add a 1 every time d0 exceeds 9. This procedure continues for all 32 bits in d1; then #$30 is added to the number in d0 to get it's CHR$ value and that value is stored as the right-most number in the buffer. If d1 is clear, then that's the end of the routine; if not, the routine repeats the whole procedure, again putting the CHR$ value in d0 as the next number in the buffer. At the end of the macro the saved registers are restored to their original value.

All of the following print macros use the CSI (Control Sequence Introduce) to print something. They start with #$9B, or 155, and are followed by a value and a case-sensitive letter. Since PRINT needs to know the length of text, I included that with each of the macros. Notice that a macro can call a storage location with the same name—A68K won't get confused. For example, if you want orange foreground, the macro ORANGEFG prints the contents of "orangefg", a four-byte string (155,3,3, and m). The macro RIGHT will move to the right a given number of spaces and then print something. Copy this file to your ASSEMBLER disk as DOSMACROS.I.

Before we use these two macro files, here are three short programs that will save you a lot of typing when you assemble and Blink your programs. Use ED, or a similar word processor, to create these script files (also included on the magazine disk):

ASB

```
.KEY NAME
        KEY NAME
A68K <NAME>.ASM
DF1:<NAME>.ASM RAM
BLINK <NAME>.O
        COPY <NAME> DF1:
DELETE <NAME>.O
```

DCOPY

```
.KEY NAME
        KEY NAME
COPY
COPYNAME>.ASMDF1:
```

DSAVE

Save each of these files in the C directory of the ASSEMBLER disk. Next, add EXECUTE to the C directory and modify the S/STARTUP-SEQUENCE by adding COPY C/ASB|DCOPY|DSAVE RAM: and COPY C/EXECUTE RAM:C/X; this also changes the EXECUTE command in RAM:C to just X. When you've finished typing your programs, you can assemble, Blink and delete the ".O" file with just one command by typing X ASB FILENAME. To save the source code and assembled program from RAM: to DF1: type

X DSAVE FILENAME; to copy an ".asm" file from DF1: to RAM: type X DCOPY FILENAME. You can modify these three programs to suit your own disk system and requirements. And those of you with more advanced systems could create your own shell commands or aliases along the same lines.

READIN' & WRITIN'

Now let's take a break and try out some of our macros. Listing 3 is a program that will let you know where the DOS Library is located and then prints two messages telling you how much available chip and fast memory you have. By using macros we will change the print styles, alter the foreground and background colors, and format the printing. The first part of the program opens the DOS Library, saves its address, executes the DOS OutPut routine and saves the conhandler address. In Part I it took ten lines to do all of that!

Next change the print style to boldface and print the "dos" message. After moving right eight spaces, return to normal print but reverse the foreground and background colors. After putting the DOS Library address in register d0, the PRINTHEXR macro will print the contents of register d0 in HEX format. Then it's back to normal printing and execution of a linefeed.

Macros can be very habit-forming.

The STYLE macro is then used to set italics type, orange foreground, and black background. After printing the "chip" message, the print reverts back to normal and the amount of chip memory is printed ten spaces to the right. The PRINTDECR macro will print the contents of register d0 as a decimal number. After a linefeed, the same procedure is repeated for "fast" memory. While this program does three times as much as the one in Part I, it still uses less lines. Try modifying the print styles and colors and see which combinations look better together. You could even create your own favorite combination macros—sort of like mixing ice cream flavors. Copy or assemble these files to the PROGRAMS disk I discussed in Part I as MEMORY.ASM and MEMORY.

& 'RITHMETIC

Up to this point all of the numbers we've been using are whole numbers. But life isn't as simple as 1+1=2. How does the Amiga handle a decimal fraction like 1.2345? It can convert all variables or numbers to a FFP (Fast Floating Point) format. There is a special MATHFFP Library that handles the math functions we'll need and, just like other libraries, has its own offsets.

All numbers can be represented as a power of 2 using logarithms—the exponent is the power that the number 2 is raised to and the mantissa is the number in front of the two. For example, the number 9 in Base 2 is 1001 or .1001*2^4, so 4 is the exponent and, ignoring the decimal, 1001 is the mantissa (FFP format always uses a mantissa starting with 1). The floating-point number is put in the 32-bit data register d0 by the SPFLT command. The mantissa is in the left-most 24 bits (3 bytes) and the exponent in the first 8 bits (7-0). The left bit in the exponent (bit 7) is used for the number's sign—0 for positive and 1 for negative. Also the value #$40 is added to all exponents. The only exception to these rules is zero—its value in FFP is always 0, in fact, 32 of them. To keep using our example, we said that the number 9 is .1001*2^4 so the mantissa is 10010000, and the exponent is #$40+4. The entire number in floating-point format would be (90)(00)(00)(44). One more example. Let's try -.0625. First that becomes -.0001 in Base 2 or -.1*2^-3. The mantissa is 10000, and the exponent is #$40-3. But since this is a negative number, we have to make bit 7 of the exponent a 1 so the entire exponent becomes #$40-3+#$80 or #$BD. This number would then be in register d0 as (10)(00)(00)(BD). Every FFP number except 0 must have at least bit 31 set. Adding or subtracting 1 from the exponent is a quick way to multiply or divide a non-0 number by two. To negate a number just reverse bit 7 (the sign); to get the absolute value clear bit 7.

MATH MACRO

Now let's discuss this new library, MATHFFP. It's opened in the same way as EXEC and DOS using the name "mathffp.library". Take a look at Listing 4—SPMATHMACROS.I. The "SP" means all computations are done with Single Precision, just as in normal Basic. The offsets for its functions have been added to the "offsets:" portion of the macro. In all cases, register d0 contains the desired number along with, if necessary, register d1 and the result will always be in d0. The math library functions are:

SPFLT—converts a whole number in d0 to a FFP number
SPFIX—converts a FFP number in d0 to a whole number
SPCMP—compare FFP numbers in d0 and d1; branch accordingly
SPTST—test the number in d0; branch accordingly
SPABS—makes the number in d0 always positive (bit 7=0)
SPNEG—multiplies the number in d0 by -1 (reverse bit 7)
SPADD—adds the FFP numbers in d0 and d1; result in d0
SPSUB—subtracts d1 from d0
SPMUL—multiply d0 and d1
SPDIV—divide d0 by d1

Many of these routines use registers d1, a0, and a1 in their computations so be sure to save any necessary data already in them. Of course you could try to program your routines to never need the data in these registers.

Also included with this macro is the MATHTRANS Library. This library handles all of the trig functions (sine, cosine, etc.) as well as other functions such as exponent, natural logs, and log10, square roots, and raising a number to a power. Again, the original value must be in d0 and the result will be returned in d0. The function SPSINCOS returns not only the sine in d0 but also the location of the cosine in d1.

There are a few new terms in this macro. NARG represents the Number of ARGuments passed by your program to the macro. NARG, by the way, in one of the few case-sensitive words used with A68K; it must be all upper-case. IFEQ is a conditional assembly meaning IF EQual to 0. So, if NARG-2 is equal to 0, there must have been two values passed to the macro and the next lines will assemble. However, if NARG-2 was not 0, the assembly would skip to ENDC (END Conditional assembly) and start assembling after there. You're not limited to IFEQ. Other combinations are:

IFNE—if not equal (to 0)
IFGT—if greater than (0)
IFGE—if greater than or equal (to 0)
IFLT—if less the (0)
IFLE —if less than or equal (to 0)

Just remember that if the condition is false, assembly will skip to the next ENDC. Copy this file to your ASSEMBLER disk as SPMATHMACROS.I. If you have enough memory you might want to modify the ASSEMBLER:S/STARTUP-SEQUENCE to copy these files into RAM: this will keep DF0: from coming on and off all the time.

Listing 5 is a math_demo that uses five of these macros—FLT, FIX, MUL, DIV, and POW. Since these are all conditional macros you can either just call the macro or pass values to it. After opening the DOS, MATHFFP, and MATHTRANS Libraries, the program next converts two numbers to FFP format, stores them in N1 and N2 and then multiplies them; although three macros were used, no values were passed to them so the conditional parts were not assembled. In the next portion, two numbers are converted to FFP format but, this time, the values N1 and N2 are passed to the DIV macro so the conditional portions will assemble. In the last portion values are passed directly to the FLT macro and N1 and N2 are passed directly the the POW macro. Having this type of capability allows for very flexible programming. In one portion you could pass numbers to a macro while in another, you can use labels. Copy or assemble these files to your PROGRAMS disk as MATH_DEMO.ASM and as MATH_DEMO.

Notice that the DOSMACROS.I file was also included in this program so you can take advantage of all of the DOS macros—PRINT, PRINTDECR, etc. Of course, if you don't want to use any of them, just omit the macro file from your "includes". If you would like to see how A68K has assembled your program, type A68K MATH_DEMO.ASM -L. This will create a new file in RAM: called MATH_DEMO.LST. Use MORE to read this file. The first portion is the three include files. The program itself starts at Line 604. You can see that all of the used macros have been assembled. Notice the difference between the MUL macro (Line 667) where no values were passed and the DIV macro (Line 773) where N1 and N2 were passed. The DOC files for A68K are included on the magazine disk so you can read some of the different assembly procedures and get more information about the program itself.

SAY WHAT?

Since we've learned a little about readin' writin' & 'rithmetic—let's teach our Amiga to speak. This will involve using the TRANSLATOR Library and NARRATOR Device. The Translator will convert the text string into phonemes and store them in an output buffer. Then the Narrator will be activated as an IO (Input/Output) device and be given the task of speaking the phonemes.

To program a device, we need to accomplish four steps related to "tasks" and "ports". Since the Amiga is multitasking, we tend to think of it as having several operations running at once but, of course, only one task can be performed at a time. All tasks therefore are either currently running, ready to run, or waiting for a specific event to occur. The programmer can also add a new task or remove an old one. Use the EXEC routine FindTask to find the task structure for a specific task by name or, by passing a 0, get the location of the current task structure in d0.

Ports are used to collect and store messages. Since the Amiga is multitasking, any task can send messages to a port, but only one of these tasks will know when a message arrives. In our example we'll want to add a port for our task with the EXEC routine AddPort by passing a port storage location defined at the end of the program. We'll also uses the

Always include the values expected as a remark alongside the macro name for reference.

EXEC routine DoIO to start the device by passing the IOrequest location.

To summarize, we'll program our device by passing a 0 to FindTask; use AddPort to create a reply port, and store the task returned by FindTask at the port address+16. Then open the device with OpenDevice, enter the required values in the IOstructure, and start the device with DoIO (the program won't continue until the task is completed) or SendIO (start the task and then continue).

Since almost everything in the Amiga is structured, let's spend some time discussing the IOrequest structure. The main advantage of structuring is that all required values are always a specific distance in bytes away from the start of the structure. This makes it easy to read or change values as long as you know their offset distance. The standard IOrequest structure looks like:

MESSAGE STRUCTURE:
0 next entry
4 previous entry
8 entry type
9 priority

10 name
14 replyport location
18 length

IO REQUEST
20 device
24 unit number
28 command (#2=read; #3=write)
30 flags
31 error status
32 number of bytes transferred
36 number of bytes to be transferred
40 data buffer
44 device offset

This is followed by the required values for the given device. The NARRATOR Device requires:

NARRATOR DEVICE (speech)
48 rate 40-400 [150] words per minute
50 pitch 65-320 [110] hertz
52 mode [natural=0], robot=1
54 sex [male=0], female=1
56 location of channel masks (3,5,10,12)
60 number of masks (4)
62 volume 0-64 [64]
64 sample frequency 5,000-28,000 [22,200]
66 mouth [0=off], 1=on
67 chanmask (assigned internally)
68 numchan (assigned internally)
69 padding

The numbers in [] are default values assigned by the device.

TRANSLATOR MACRO

Listing 6 is the TRANSLATORMACROS.I file that does most of the work for us. Since using the TRANSLATOR Library means we will also need the NARRATOR Device., I've included the set-up necessary for both in this file. There is only one offset in the library—TranslateText. This routine requires the address of the string to be translated in a0, the string length in d0, the buffer location where the translated phonemes will be stored in a1 and that buffer size in d1.

Next are the various equates. The two device commands Read and Write are defined, followed by the various speech parameters. The first macro OPENNARDEVICE opens the device by putting the device name in a0, the IOrequest in a1, the unit number (0) in d0, and flag requirements (0) in d1. If the device can't be opened, d0 will contain an error message number and the program must terminate.

The other macro SAY will put all of the speech parameters into their proper locations, compute the text string length, translate the text into phonemes, and speak them. The values for speech must be in the following order—text, rate, pitch, mode, sex, volume, and sample frequency. They must be in this order, but, since the macro will conditionally assemble them, you may omit any setting being sure to keep the ",". For example, you could use SAY TEXT1,160,,ROBOT,,,20000. If you want to use only the default settings, just pass the text location; NARG will equal 1 and the macro will conditionally assemble. Now you can see why I recommended keeping a record of all macros and what they require. Since various storage areas are needed, it makes sense to include some of them in the macro; the program doesn't care where they are located. This keeps you from having to remember a lot of extraneous material and lets you concentrate on the program, so I've reserved space for the IOrequest structure in the file. If you have a very long text string, you may need to increase the buffer from its present size of 512 bytes. Copy this file to your ASSEMBLER disk as TRANSLATORMACROS.I.

Our final program, Listing 7, is rather short, thanks to all of the work done by the macros. The program will speak three text phrases using different voice settings. After opening the TRANSLATOR Library, the IOrequest is set up for the NARRATOR Device. Then each text phrase is spoken using the values passed in the SAY macro. At the end of the program the port is removed, the device closed, and the libraries closed. Then space is reserved for the stack pointer, library and device locations, the port, the two text strings and their phonemes buffers, and the channel masks. Notice that the length is computed using "*" which means "this location". Once the initial set-up is out of the way you could use these macros to add speech to any program. And if your text string was already converted to phonemes you wouldn't need the TRANSLATOR Library, but that's a lot of extra work. Copy or assemble these files to your PROGRAMS disk as TALK.ASM and TALK.

BACK TO THE FUTURE

Again, we've covered a lot of ground but I think that the macro files will make programming easier for you. Feel free to modify these files in any way you want. Be sure, however, to include these files along with the source code for any programs you write for someone else or have published. In the next article I'll discuss the INTUITION and GFX Libraries and we'll write some graphic programs using arrays and a quick PSET routine.

The listings and all necessary files for Programming the Amiga in Assembly Language can be found on the AC's TECH Disk.

Please write to:
William P. Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Affordable Excellence

ReSource macro disassembler — NEW VERSION!

ReSource V5 is an intelligent interactive disassembler for the Amiga programmer. **ReSource V5** is *blindingly* fast, disassembling literally hundreds of thousands of lines per minute from executable files, binary files, disk tracks, or directly from memory. Full use is made of the Amiga windowing environment, and there are over 900 functions to make disassembling code easier and more thorough than its ever been.

Virtually all V2.0 Amiga symbol bases are available at the touch of a key. In addition, you may create your own symbol bases. Base-relative addressing, *using any address register*, is supported for disassembling compiled programs. All Amiga hunk types are supported for code scan.

ReSource V5 runs on *any* 680x0 CPU, but automatically detects the presence of an 020/030 CPU and runs *faster* routines if possible. **ReSource V5** understands 68030 instructions and supports the new M68000 Family assembly language syntax as specified by Motorola for the new addressing modes used on the 020/030 processors. **ReSource V5** and **Macro68** are among the few Amiga programs now available that provide this support. Old syntax is also supported as a user option.

An all new online help facility featuring hypertext word indexing is included. This enables you to get in-depth help about any function at the touch of a key! **ReSource V5** includes a new, completely rewritten manual featuring two tutorials on disasssembly, and comprehensive instructions for utilizing the power in **ReSource V5.**

ReSource V5 will enable you to explore the Amiga. Find out how your favorite program works. Fix bugs in executables. Examine your own compiled code.

"If you're serious about disassembling code, look no further!"

ReSource V5 requires V1.3 or later of the Amiga OS, and at least 1 megabyte of ram. **ReSource V5** supercedes all previous versions.

Suggested retail price: US$150

Buy Macro68 and ReSource together and get $30 off!

Macro68 macro assembler — NEW VERSION!

Macro68 is the *most* powerful assembler for the entire line of Amiga personal computers.

Macro68 supports the entire Motorola M68000 Family including the MC68030 and MC68040 CPUs, MC68881 and MC68882 FPUs and MC68851 MMU. The Amiga Copper is also supported, eliminating the need for tedious hand coding of 'Copper Lists'.

This fast, multi-pass assembler supports the new Motorola M68000 Family assembly language syntax, and comes with a utility to convert old-style syntax source code painlessly. The new syntax was developed by Motorola specifically to support the addressing capabilities of the new generation of CPUs. Old-style syntax is also supported, at slightly reduced assembly speeds.

Most features of **Macro68** are limited only by available memory. It also boasts macro power unparalleled in products of this class. There are many new and innovative assembler directives. For instance, a special structure offset directive assures maximum compatibility with the Amiga's interface conventions. A frame offset directive makes dealing with stack storage easy. Both forward and backward branches, as well as many other instructions, may be optimized by a sophisticated N-pass optimizer. Full listing control, including cross-referenced listings, is standard. A user-accessible file provides the ability to customize directives and run-time messages from the assembler.

Macro68 is fully re-entrant, and may be made resident. An AREXX™ interface provides "real-time" communication with the editor of your choice. A number of directives enable **Macro68** to communicate with AmigaDos™. External programs may be invoked on either pass, and the results interpreted. Possibly the most unique feature of **Macro68** is the use of a shared-library, which allows resident preassembled include files for incredibly fast assemblies.

Macro68 is compatible with the directives used by most popular assemblers. Output file formats include executable object, linkable object, binary image, and Motorola S records. **Macro68** requires at least 1 meg of memory.

Suggested retail price: US$150

The Puzzle Factory, Inc.
P.O. Box 986, Veneta, OR 97487
"Quality software tools for the Amiga"
For more information, call today! Dealer inquires invited.
Orders: (800) 828-9952
Customer Service: (503) 935-3709

Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc.

VISA / MasterCard

Check or money order accepted
no CODs

Circle 102 on Reader Service Card

Amiga Voice Recognition

by Richard Horne

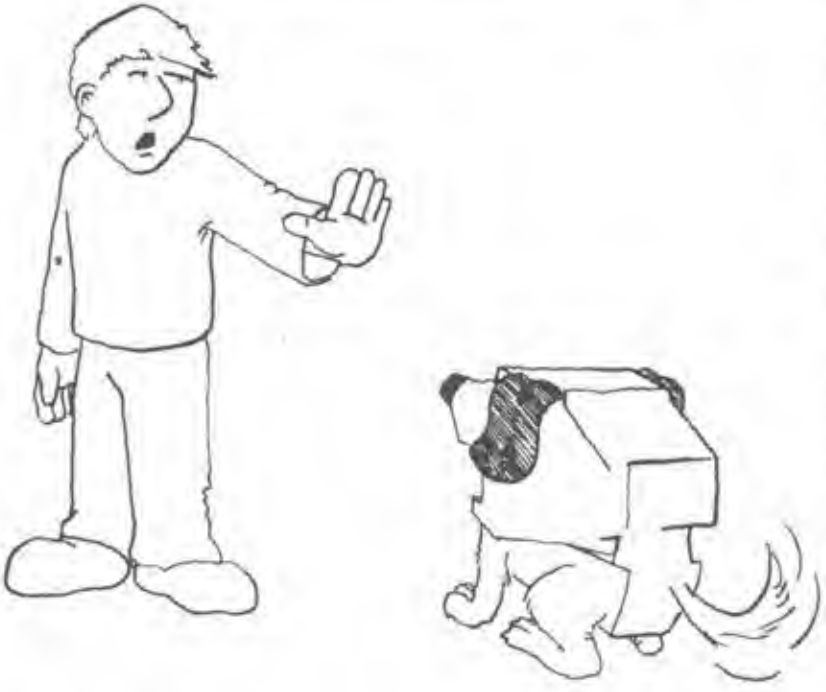
One of the most interesting and difficult tasks to be undertaken by modern computer science is that of recognition of human speech. Special purpose (and expensive) hardware is available commercially that provides fixed vocabulary speech recognition in real time. Now there is available a public domain disk-based Amiga library that will permit Amiga programmers to add voice recognition to their applications using the Perfect Sound 3 audio digitizer. This voice.library provides functions for learning and recognition of user-defined words or phrases in near real time using any Amiga computer.

How Can a Computer Recognize Speech?

Two primary methods of computer voice recognition have been studied extensively. These can be called speaker independent and speaker dependent voice recognition. Both operate on digitized samples of human speech and attempt to match received speech samples against characteristics of a fixed vocabulary.

Speaker-independent voice recognition is the more general approach. This technique requires the computer to analyze a received speech sample against statistical vocabulary characteristics derived from a large population of individuals. This technique can in theory recognize the same word or phrase spoken by different individuals and account for differences in speech accent, pitch, and speed. In practice, this technique requires so much processing power and time that it is not practical for use with most desktop computers.

Speaker-dependent voice recognition is more practical for our use. This technique requires that the individual user provide digital samples of his/her voice for each word or phrase of the desired vocabulary. The characteristics of these samples are stored in memory. Characteristics of incoming digitized words or phrases are then compared against these to find the best match. The disadvantage of this approach is that only the speech of one individual will be recognized. The advantage is that the technique is relatively fast and accurate.



What Is Voice.library?

Included on disk is a copy of Amiga voice.library as well as VoiceLibrary.doc which documents each of the library functions. If voice.library is copied into your sys:libs directory, your programs will be able to access its functions to easily implement speaker-dependent voice recognition.

As might be expected, the primary functions contained in the library to assist in voice recognition are the "Learn" function which samples and stores characteristics of the user's voice for each vocabulary word or phrase, and the "Recognize" function which compares characteristics of incoming digitized speech against the learned vocabulary to produce a match. The principles of operation of these primary functions are explained below.

Learning a Word

In order to learn, or store characteristics of vocabulary words or phrases, a decision must be made as to what characteristic of speech is to be used for recognition. Many different techniques have been tried including phoneme-based systems which attempt to divide incoming words into distinct and unique sounds (phonemes), as well as systems which examine the frequency and amplitude of the incoming signal as a function of time. For simplicity and speed, the technique used by voice.library is to determine the frequency content of the spoken word or phrase over a period of time. This frequency spectrum information is stored in memory as a frequency map for

later use in recognition of incoming words. The learned vocabulary consists of a stored sequence of these frequency maps, each associated with one word or phrase of the vocabulary.

In constructing a frequency map, tradeoffs are necessary between frequency resolution and time span allowed for each word or phrase. We need the highest resolution frequency analysis available that is consistent with the time scale of the incoming word or phrase and that can be computed in near real time on a standard Amiga. The compromise chosen for use with voice.library is to compute a frequency analysis at 72 points in time over a span of 3/4 of a second. For each of these 72 points, 32 frequency data points are computed corresponding to frequency content in 32, 100 Hz bands from 0 Hz to 3200 Hz. This compromise limits the length of learned words to 3/4 second, but this is adequate for all but the very longest words. Also, only frequency data below 3200 Hz is utilized directly in voice recognition. However, the strongest frequency components of human speech are generally below this limit.

Before calling the "Learn" function, the user must decide on a sequence and number of vocabulary words or phrases to be used. Then each vocabulary entry is learned according to the following synopsis:

```
MapAddress = Learn (MapBuffer,
Text, Screen, Sequence, X, Y)
    d0                  a0
a1    a2      d0          d1 d2
```

Each frequency map is made up of 72 long words of data—each representing 32 frequency points—plus a 16-byte header for the associated ASCII text (304 bytes total). "Learn" requires the user to reserve a MapBuffer in memory equal to the size of vocabulary desired (number of words or phrases) times 304 bytes. The MapBuffer address is passed to "Learn" in a0. Address of a null terminated text string representing the word or phrase to be learned is passed to "Learn" in a1.

The "Learn" function will open its own window on the screen specified in a2 at a position X, Y specified in d1 and d2. The user will then be prompted to speak the specified word or phrase to obtain three good digital samples. Internally, these three samples are analyzed for frequency content and transformed into a frequency map (304 bytes) which is stored in the MapBuffer in order, according to the sequence number specified in d0. "Learn" returns the memory address within MapBuffer at which this particular frequency map is stored.

"Learn" is called separately for each word or phrase in the vocabulary. After every word or phrase has been learned, MapBuffer will be filled with a complete sequence of frequency maps for later use in voice recognition. See voicelibrary.doc on disk for a complete description of this function.

In constructing a frequency map, tradeoffs are necessary between frequency resolution and time span allowed for each word or phrase.

Word Recognition

Recognizing a word requires exactly the same process as learning a word. That is, a frequency map of 3/4 seconds of time is computed for incoming words. Then, using a pattern recognition algorithm, this incoming map is compared with every learned map in the vocabulary. A synopsis of the "Recognize" function is as follows:

```
SequenceNum = Recognize (MapBuffer, Vocabulary, Resolution)
d0                      a0          d0          d1
```

MapBuffer contains a sequence of frequency maps produced by "Learn" corresponding to each word or phrase in the vocabulary. Mapbuffer address is passed to "Recognize" in a0. Number of words or phrases in the vocabulary are passed to "Recognize" in d0.

"Recognize" listens for an incoming word, computes its frequency map, and compares this map to the sequence of maps contained in MapBuffer. The Sequence Number of the word or phrase in MapBuffer which is most similar to that of the incoming word is returned in d0. Note that the number "0" represents the first word, "1" the second, and so on. See voicelibrary.doc on disk for a complete description of this function as well as various errors that might be encountered.

"Recognize" will operate at either high resolution (d1 = 0) or low resolution (d1 = 1). High resolution computes a frequency analysis of the incoming word or phrase at 72 points in 3/4 second whereas low resolution computes only 36 points in 3/4 second. High resolution is somewhat better at word recognition, but takes almost twice the processing time.

Multitasking Word Recognition

In many applications it will be convenient to have the "Recognize" function operate in the background as a separate task under the Amiga's multitasking operating system. Voice.library provides this capability with the "AddVoiceTask" function. A synopsis of "AddVoiceTask is as follows:

```
AddVoiceTask (MapBuffer, MsgPort, Vocabulary, Resolution)
              a0          a1       d0          d1
```

"AddVoiceTask" is similar in function to "Recognize" except that here, a separate task is started which listens for incoming words or phrases and returns messages to the user's Message Port indicating the Sequence Number of the frequency map in Mapbuffer which best matches the frequency map of the incoming word. MapBuffer address and Message Port address are passed to "AddVoiceTask" in a0 and a1. Number of words or phrases in the vocabulary are passed to "AddVoiceTask" in d0.

The messages sent to Message Port are designed to mimic IDCMP messages with im_Class = $0. Thus you can receive and process these messages at either an Intuition window IDCMP message port or at a custom message port of your own. The im_Code for these messages consists of the sequence number of the frequency map in MapBuffer that best matches the frequency map of the incoming word or phrase. See voicelibrary.doc on disk for a complete description of this function and the various errors that might be encountered.

A Pickle, Anyone?

The VoiceDemo program on disk gives an example of the use of voice.library for learning and recognition of words. Entire assembly source code for this demonstration is contained in VoiceDemo.asm. This program is a simple illustration of the use of voice.library to create and recognize a vocabulary consisting of the six phrases, "Peter... Piper... Picked... A Peck Of... Pickled... Peppers." VoiceDemo uses the voice.library "Learn" function to store a frequency map of each phrase, and the "AddVoiceTask" function to listen for and recognize incoming phrases. The Amiga Utilities "Say" function is used to repeat the phrase back to the user. A plot of the frequency map of the incoming phrase is also displayed. Key sections of the assembly code are described below.

Two primary methods of voice recognition have been studied: speaker dependent and speaker independent.

First, library offset values must be defined for voice.library functions. These values are defined for all functions in voicelibrary.doc on disk. Offsets for the primary functions used in VoiceDemo are defined as follows:

Access to voice.library is established using the Exec "OpenLibrary" function. Then library functions may be called as subroutines using these offsets from the library base address.

To define the desired vocabulary, I have listed each phrase of the vocabulary in order in a "WordList." Then each phrase of the vocabulary is assigned in order to menu items making up a complete "Learn" menu. When the user selects a vocabulary phrase from the "Learn" menu, VoiceDemo executes the following routine:

MapBuffer consists of an area of memory equal to the number of vocabulary entries (six) times the size of each frequency map (304 bytes). The sequence number of the word to be learned is defined by the number of the menu item from the "Learn" menu selected by the user (ItemNumber). InfoScr is the screen address of the background information screen on which the "Learn" window will open. The position of the "Learn" window is specified to be X=130, Y=40.

This Learn Routine is called for each phrase in the vocabulary by user selection of the menu item corresponding to each phrase. After all phrases have been learned, MapBuffer is filled in order with a sequence of six frequency maps corresponding to each of the six phrases of the vocabulary.

Next, the user may select "Recognize Word" from the project menu. VoiceDemo will open a small "Voice Window" having an IDCMP message port and then call the voice.library "AddVoiceTask" function to recognize incoming words and send messages to the message port when a word or phrase is recognized. A partial listing of this routine is as follows:

RecognizeRoutine:

VoiceWdw is the Intuition Window definition (including an IDCMP message port at wd_UserPort) for receiving messages

when words are recognized. MapBuffer contains the frequency maps previously learned. Resolution has previously been defined by user menu selection as "0" for high resolution or "1" for low resolution frequency mapping.

When "AddVoiceTask" is called, background voice recognition begins. VoiceDemo may then "Wait" for messages to be received at the message port of VoiceWdw. Note that either Voice messages or Intuition messages may be received at the same message port. These can be separated upon receipt by MessageClass. Only Voice messages have MessageClass = $0. MessageCode equals the sequence number of the recognized word based on the order in which word frequency maps have been learned and stored in MapBuffer.

VoiceDemo proceeds from this point to plot the frequency map of recognized words in the VoiceWdw and to repeat the recognized phrase back to the user using the "Say" routine. See the complete listing VoiceDemo.asm on disk for further details. The frequency map plot is meant to graphically illustrate the structure of this map as used for voice recognition. Each map consists of 72 vertical lines (36 in low resolution). Each vertical line is made up of 32 points representing frequency content in 32, 100Hz frequency bands on the vertical axis. The horizontal axis represents 3/4 second of time. This plot is in effect a low resolution voice print of the recognized word or phrase.

Practical Considerations

Choice of vocabulary words or phrases is important to successful use of voice.library. Words that sound distinctly different from each other will be more reliably recognized. If the words "tension" and "pension" are in your vocabulary, "Recognize" will frequently confuse the two.

Even with distinctly different words, you must be careful to pronounce all syllables of each word distinctly and forcefully. For those of you with children, speak as if you are telling your offspring for the third time to "CLEAN UP YOUR ROOM!" This is about the proper level of distinct and forceful speech.

"Learn" and "Recognize" are sensitive to any change in the pitch or rate of your speech. If you have a cold, or are under stress, such as when demonstrating your amazing program to a dubious mate, changes in the pitch or rate of your voice may confound voice recognition. It may be necessary to relearn difficult words under these conditions.

Voice recognition will be confused by background noise. Humans can frequently tune out distracting background noise, but voice.library cannot be that sophisticated. If Bart Simpson's voice is coming from a television in the background, don't be surprised at the results.

Microphone selection can be important. Since frequency mapping in voice.library only uses a frequency band of 3200 Hz, a super high bandwidth microphone is not necessary or desirable. It can be helpful to have a microphone with a push button on-off switch. A microphone that is active all the time will sometimes trigger the "Recognize" function with stray noise rather than with the start of a spoken word.

And finally, limit your vocabulary to the minimum required. The maximum capacity of voice.library is 64 words or phrases. However, recognition time increases with vocabulary size, not to mention the time required to "Learn" every vocabulary entry.

Conclusion

Voice.library is the result of several years of related work including the earlier public domain program "Amiga Spectrogram" for frequency analysis of audio data. Voice.library was written to demonstrate the power of the Amiga and to encourage programmers to develop even more interesting applications of Amiga audio. The author will be happy to answer any questions or discuss your ideas for use of voice.library. He can be reached on-line as follows:

COMPUSERVE	71777,407
GENIE	RHORNE
PORTAL	RHorne

Your comments or suggestions for future improvement are also welcome.

Please write to:
Richard Horne
c o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Dynamically Allocated Arrays

By Charles Rankin

There I was in the depths of writing a program and realized I needed to use some two-dimensional arrays. No problem, I thought. Then I realized that I needed to be able to allocate these arrays to any dimension—on the fly—at runtime. Uh, oh. Now I had a problem. I dug out my trusty C references and began looking. "Won't be long now," I thought. Needless to say, after many hours of searching I didn't find anything. So I went down to the local bookstore to examine a few more books. Still nothing. It was obvious that if this was going to work, I was going to have to make it work. So I sat down with my handy-dandy C compiler, a pencil, paper, a tall glass of cola and some chips and came up with the technique below.

Everybody knows about arrays, but not everybody knows that you don't necessarily have to specify their size when you compile. Take, for example, these two one-dimensional cases:

```
A) 1-dimension: hard coded array size.
        #include <stdio.h>
                void main() {
                int square[9];  /* declare array of size 9 */
                int i;      /* counter */
                for (i = 0; i < 9; ++i)
                 square[i] = i * i;     /* calculate square
*/
for (i = 0; i < 9; ++i)
                 printf("%d\n",square[i]); /* output array */
  } /* end main */

B) 1-dimension: dynamic array size.
#include <stdio.h>
void main() {
        int *square = NULL;  /* declare pointer for our
array*/  int i;        /* counter */
        /* allocate memory for our array */
        if ((square = (int *)malloc(9 * sizeof(int)))==NULL)
        {
          printf("Could not allocate memory\n");
          exit(0);
        }
        for (i = 0; i < 9; ++i)
         square[i] = i * i;     /*calculate square */
         for (i = 0; i < 9; ++i)
         printf("%d\n",square[i]); /* output array */
free(square);  /* free allocated memory */
} /* end main */
```

Both of these programs will produce the same output: a list of the squares of the numbers 0 to 8. The first piece of code is very straightforward. The second piece of code may need a little bit of explanation. The call to malloc() allocates enough memory to hold 9 integer variables. The variable, square, holds the pointer to this information. It may seem strange that we are accessing the information held by this pointer with what appears to be an array access. To understand this, you need to realize that the compiler translates the array reference square[i] into the pointer reference *(square + i). Thus any array reference is translated to a pointer offset into the memory we allocated.

Things are a little different when you get into two dimensions. A little more setup is required to achieve the same goal. The following pieces of code show a two-dimensional example.

```
A) 2-dimensions: hard coded array size
 #include <stdio.h>
 void main() {
   int sum[10][10]; /* declare 10 X 10 array */
   int i, j;        /* counters */
   for (i = 0; i < 10; ++i)
    for (j = 0; j < 10; ++j)
     sum[i][j] = i * j; /* fill array */
   for (i = 0; i < 10; ++i) {  /* loop through array */
for (j = 0; j < 10; ++j)
```

Everybody knows about arrays but not everyone knows you don't have to specify their size when you compile.

This program initializes a 10 X 10 array. It then stores in element sumsquare[row][column] the sum of row squared and column squared. As before, the first piece of code is very straightforward. The initialization of the second piece of code is a little harder to see. First we allocate enough room for 100 integers and 10 integer pointers. Then what essentially happens is we partition the 100 integers into 10 separate 10 integer groups and point one of the integer pointers to each of the groups. This way when we access sumsquare[i][j], the sumsquare[i] part points to one of the integer groups and the [j] part references a specific integer within the group. In a concise manner the array reference sumsquare[i][j] is converted into the pointer reference *(*(sumsquare + i) + j).

I think another small example would help to understand the initialization process. Take this 3 X 3 example below. Let's call the array test (actually test is an int **). It may help to refer to the code of section B) above with the array reference sumsquare changed to our new variable test.

Note that a "-" in the contents field represents unknown contents. These contents are unknown because we don't know what was in the memory allocated by malloc.

```
ADDR CONTENTS  ---  ----
0    [ 3 ]  test[0] = &test[3]
1    [ 6 ]  test[1] = &test[0][3]
2    [ 9 ]  test[2] = &test[1][3]
3    [ - ]  This would correspond to test[0][0]
4    [ - ]   "   "   "   " test[0][1]
5    [ - ]   "   "   "   " test[0][2]
6    [ - ]   "   "   "   " test[1][0]
7    [ - ]   "   "   "   " test[1][1]
8    [ - ]   "   "   "   " test[1][2]
9    [ - ]   "   "   "   " test[2][0]
10   [ - ]   "   "   "   " test[2][1]
11   [ - ]   "   "   "   " test[2][2]
```

The first part of the allocated memory contains the row pointers into each row of data in memory. Note that since we were using a 3 X 3 array we use only indices 0 through 2 when referencing array locations. So why then did we access test[3] at all (on the address 0 line above)? The reason is that this is actually the first part of our data storage. It is the first location after our 3 row pointers. And in reality we wanted only the address of this memory location so that we could point row zero at it; this is the reason for the & (address) operator. The same argument applies for the access of test[0][3]. The test[0] portion is a pointer to the actual data. Looking at test[0], we see that the address is 3. Then the [3] portion (of test[0][3]) takes us 3 places farther into memory (or address 6). This is where data for row 1 will start—remember we start with row 0. Thus we need the address of it, which again explains the & operator).

We then initialize the row 2 pointer with test[2] = &test[1][3], which sets data storage for row 2 at address 9 (test[1] = 6, test[1][3] goes 3 further, or address 9). We have now finished all our initialization and the array can be used like any other. This correlates to completing the for loop in the code of section B) above.

The above examples lead to a procedure which will allow you to allocate a two-dimensional array of any size that you wish. See Listing 1 for this procedure. Note that Listing 1 is specifically for arrays of integers. You could define this procedure for any type of variable also, be it float, double, short, char, or whatever. It would also be possible to define a macro that would allow you to define any type of array at any time (Listing 2). The technique described here is also amenable to arrays of 3 or more dimensions; one must simply initialize another level of pointers. Also, take careful note that the array indices still start at 0 as they do in regular arrays.

I believe this technique is very beneficial in any program that uses arrays. If the user has only small requirements for the program then he does not have to be burdened with the extra memory required by the hard-coded arrays that the programmer used. On the other hand, it allows users with big demands and lots of memory to use that memory instead of being confined by the hard-coded array sizes the programmer used. I hope some of you out there can benefit from this technique as much as I have.

Listing 1

```
/*************************************************************
/* A simple program that asks the user for the dimensions    */
/* of a 2 dimensional array and then prints out the sum      */
/* of the row squared and the column squared for each        */
/* element.                                                  */
/*                                                           */
/* Illustrates the use of the procedure for dynamically      */
/* allocating arrays.                                        */
/* Author: Charles Rankin                                    */
/* Compiles with Lattice C v5.05 (lc -L program name)        */
/*************************************************************

#include<stdio.h>
/*****************************************************
* init_array - procedure to allocate memory for a    *
/*              2-dimensional array and initialize    *
/*              the pointers.                         */
/* NOTE: even though the array is of size y by x,     *
/*       the indices are still only 0 to (y-1)        *
/*       and 0 to (x-1).                              *
/*****************************************************

int **init_array(y, x)
int y,x;     /* the # of rows and columns respectively */
{
int **array;  /* pointer to the 2-dimensional array */
int i;        /* dummy loop variable */

/* if we don't get the memory return NULL */
if (array = (int **)malloc(y * (y * sizeof(int *)
 +(sizeof(int *)) == NULL) return(NULL);

/* else initialize the pointers */
else {

 array[0] = (int *)&array[y];
 for(i = 1; i < y; i++)
   array[i] = (int *)&array[i-1][x];

 return(array);

 } /* end of the if statement */
} /* end of procedure init_array */

void main()
{
 int **square;  /* pointer to our array */
 int row, y;          /* how many rows in array */
 int column, x;       /* how many columns in array */
 int i,j;             /* dummy loop variables */
 while (1)
 {
  printf("\nPlease enter the number of rows in"
   " the array (0 to quit): ");
  scanf("%d", &row);
  if (row == 0)       /* user wanted to quit */
   {printf("\n Terminated from\n\n");
   exit(0);
  }
  printf("\nPlease enter the number of columns in"
   " the array: ");
  scanf("%d", &column);

  /* Until terminated by memory */
  if (square = init_array(row, column)) != NULL)
  /* call procedure to initialize array */
  {  for(i = 0; i < row; i++)
     {
        for(j = 0; j < column; j++)
        {
          /* calculate sum of squares */
          square[i][j] = i*i + j*j;
          printf("%4d ", square[i][j]);  /* output */
        } /* end j loop */
```

```
        printf("\n");
        } /* end i loop */
    }
    else printf("Could not allocate memory for array!\n");
  } /* end while */
} /* end main */
```

Listing 2

```
/*****************************************************************
/*A simple program that asks the user for the dimensions    */
/* of a 2 dimensional array and then prints out the sum      */
/* of the row squared and the column squared for each        */
/* element.                                                  */
/*                                                           */
/* Illustrates the use of a macro for dynamically            */
/* allocating arrays.                                        */
/*                                                           */
/* Author: Charles Rankin                                    */
/* Compiles with Borland C v3.05 (lc -l powrow-cases)        */
/*****************************************************************/

#include <stdio.h>
 /* This is the macro that does all the work. It    */
 /* initializes the 2 dimensional array "name" of   */
 /* variable type "type" with dimensions "x" & "y"  */
 /* Note that "x" and "y" could even be expressions */
 /* such as (rows + 1) or something like that.      */
 /* Also, "type" can be just about any type you     */
 /* want it to be, like short, long, unsigned char, */
 /* float, or even int *.                           */

#define INIT_ARRAY(name,type,x,y) {
 name = (type **)malloc( (y) * (x) * sizeof(type) +
  sizeof(type *)); \
 if (name != NULL) for (i, name[0] = (type *)&name[x]; \
 i<(x); i++) name[i] = (type *)&name[i-1][(x)]; }
void main()
{
 int **square = NULL;     /* pointer to our array */
 int row = 0;             /* how many rows in array */
 int column = 0;          /* how many columns in array */
 int i,j;                 /* dummy loop variables */

 while (1) {
  printf("\nPlease enter the number of rows in"
   " the array (0 to quit): ");
  scanf("%d", &row);
  if (row == 0) {         /* user wanted to quit */
   if (square) free(square);
   exit(0);
  }
  printf("Please enter the number of columns in"
   " the array: ");
  scanf("%d", &column);
  /* don't reallocate memory */
  if (square) free(square);

  /* call procedure to initialize array */
  INIT_ARRAY(square, int, row, column);
  if (square != NULL) {

    for(i = 0; i < row; i++) {

     for(j = 0; j < column; j++) {

      /* calculate sum of squares */
     sqrArea[i][j] = (i*i) + (j*j);
     printf("%4d ", sqrArea[i][j]);   /* output */

     } /* end j loop */

     printf("\n");

    } /* end i loop */
  }
  else printf("Could not allocate memory for array!\n");

 } /* end while */

} /* end main */
```

Please write to Charles Rankin
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Should You?

YES!

Amaze Them Every Month!

Amazing Computing For The Commodore Amiga is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing* .

A Guide For Every Amiga User.

Give the Amiga user on your gift list even more information with a **SuperSub** containing *Amazing Computing* and the world famous ***AC's GUIDE To The Commodore Amiga***. *AC's GUIDE* (published twice each year) is a complete listing of every piece of hardware and software available for the Amiga. This vast reference to the Commodore Amiga is divided and cross referenced to provide accurate and immediate information on every product for the Amiga. Aside from the thousands of hardware and software products available, *AC's GUIDE* also contains a thorough list and index to the complete Fred Fish Collection as well as hundreds of other freely redistributable software programs. No Amiga library should be without the latest AC's GUIDE

More TECH!

AC's TECH For The Commodore Amiga is an Amiga users ultimate technical magazine. *AC's TECH* carries programming and hardware techniques too large or involved to fit in *Amazing Computing*. Each quarterly issue comes complete with a companion disk and is a must for Amiga users who are seriously involved in understanding how the Amiga works. With hardware projects such as creating your own grey scale digitizer and software tutorials such as producing a ray tracing program, *AC's TECH* is the publication for readers to harness their Amiga to fulfill their dreams.

To order phone
1-800-345-3360
(in the U.S. or Canada)
Foreign orders:
1-508-678-4200
or
FAX 1-508-675-6002.
or
CLIP THIS COUPON AND MAIL IT TODAY!

MAIL TO:
Amazing Computing
P.O. Box 2140
Fall River, MA 02722-2140

YES! *The "Amazing" AC publications give me* ***3 GREAT*** *reasons to save! Please begin the subscription(s) indicated below immediately!*

Name _____

Address _____

City _____ State _____ ZIP _____

Charge my ☐ Visa ☐ MC # _____

Expiration Date _____ Signature _____

Please circle to indicate this is a **New Subscription** or a **Renewal**

1 year of **AC**	12 big issues of Amazing Computing! Save over 40% off the cover price!	US $27.00 ☐ Canada/Mexico $34.00 ☐ Foreign Surface $44.00 ☐
1-year SuperSub	AC + AC's GUIDE – 14 issues total! Save more than 45% off the cover prices!	US $37.00 ☐ Canada/Mexico $54.00 ☐ Foreign Surface $64.00 ☐
2 years of **AC**	24 big issues! Save over 59%! US only.	US $41.00 ☐
2-year SuperSub	28 big issues! Save more than 56%! US only.	US $59.00 ☐
1 year of **AC's TECH**	4 big issues! Limited time offer – US only!	US $43.95 ☐

Please call for all other Canada/Mexico/foreign surface & Air Mail rates.
Check or money order payments must be in US funds drawn on a US bank; subject to *applicable* sales tax.

Implementing an ARexx Interface in your C Program - Part 2

by David Blackwell

Since we got most of the hard work done in the first article, this should be short and sweet. We left off with a functioning program with only a minimal ARexx interface. The program, while a completely functioning, was not very functional. Well, even after we are finished with our ARexx interface, the program will still be barely functional. However, the purpose of this article was to introduce you to what is required to add an ARexx interface to your program. If you continue with the program, adding improvements and ending up with an excellent utility, then you have received a bonus. That is up to you.

As promised in the last article, I continued my testing of the rexxapp.library to see why the SendRexxCmd did not appear to execute synchronously as implied by the document file included with the library. During my test, I also discovered that the SyncRexxCmd exhibited the same behavior.

In true synchronous execution, the function called should not return until all processing is complete. This just was not the case with the two previously mentioned functions. They seemed to return immediately, causing problems in my code following the function call since it was expecting to use the results returned by the functions.

Jeff Glatt of Dissidents Software, author of the rexxapp.library, was kind enough to provide me with a copy of the library source code. This answered all the questions I had about the routines. They package the request in an ARexx message and send it off and then return control back to the calling program. When the ARexx message returns, these routines call another function—the function you provide in the case of the SendRexxCmd and an internal function in the case of the SyncRexxCmd—and consider this the synchronous portion of their execution. I consider this pseudo-synchronous at best, as your program will have to be prepared to continue execution directly after the function call without the necessary return values you need to continue your work. This is essentially the same as asynchronous execution.

Back to the Interface

I also promised to add more commands to the RexxData structure and I have again kept my promise. I added a total of eight commands. Three of these commands were already available in the host application. I just made them accessible through the ARexx interface. Three other commands are available only through the ARexx interface. These commands are used to provide information to the ARexx macros about the currently active file. This information is constantly available to the host application through global variables; therefore, the host application has no need to call these functions. Only two new commands were added to the program that are called by both ARexx macros and the host application.

These two new commands allow you to add entries to the database files created by the save command and to view selected entries in the files by searching the key fields in the file. I added these commands using two completely different approaches. The first way shows you how you can add

commands to your program by adding a function to the host application and then making it accessible through the ARexx interface. The second method uses an ARexx macro to provide a new command and then shows how to access it in the host application.

Adding Commands to the Host Application

The first method is really nothing new; you add a function to the host application as you would any other program. You add a new menu selection, update your switch code to receive the Intuition messages pertaining to the new menu selection and add the function(s) needed to implement the new command. To make it accessible through the ARexx interface, add the command name to the RexxData structure, add one to the NUMCMDS label and add the rexx routine to call the new function. I will demonstrate how to call a function in the host application later when I discuss modifications I made to the dispatch function.

ARexx Macros as Commands

First you might ask, isn't that what you normally do with ARexx macros? To that question I would have to answer, yes; so I will. That is what you do—sort of. You can use ARexx macros to extend the capabilities of a program by adding commands the host application does not have. This is not exactly what I am doing though. There is one subtle difference. The ARexx macros I am adding are intended to be part of the host application. They are being used as disk-resident commands rather than memory resident commands. See the difference? No. Well, I will explain it.

By coding into the host program, all the menu selections for the commands and providing a way for the host program to call the ARexx macros, you make the macros an integral part of the program and not simply an enhancement. In this fashion, you create a standard set of macros that can easily be enhanced or embellished as the user sees fit. The main advantage I am looking for is the reduction in the size of the host application.

If the host application is just a small core of essential routines, while the remainder of the commands are disk resident, the program can be quite small. It would mostly be the small routines required to call the ARexx macros into action. The ARexx macros and the host application could then work together, passing the necessary information back and forth to each other to get the overall task accomplished.

Using the rexxapp.library presented a little challenge when I went to add an ARexx macro to work in this manner. I wanted to use the ASyncRexxCmd or the SyncRexxCmd but you need a pointer to an ARexx message to pass to these commands. When the command is generated by a menu selection, there is no message pointer to pass to these functions. I had to find another way to accomplish this.

What I worked out was to add another message port to my program for asynchronous communications. I then added the signal for the new asynchronous message port to the variable I use in my main function to wait on for input events. I added a routine to handle activity at this message port. Finally, I added a small routine to create an ARexx message and put an Argstring containing the name of the command I wanted to execute in the ARG0 field of the message structure. I then send this message to my own ARexx message port. That is a real roundabout way to execute an internal command.

Dispatch Routine

I also modified the dispatch routine to allow passing the arguments supplied to it by the rexxapp.library on to the functions it calls. You will notice that I had to rename all the function names in the RexxData structure to accomplish this. Here is an example:

{"open", (APTR)&openfile} is now

{"open", (APTR)&rexxopenfile},

And instead of calling the openfile function directly, it is now accomplished like this:

```
BOOL rexxopenfile(struct RexxMsg *msg, char *str, struct
RexxData *data)
{       return(openfile());     }
```

This change was necessary, because we will want some of the commands in our command-associated list to be able to execute macros. This is so because all the commands in our command associated list are called in exactly the same manner. So if one of these commands needs certain arguments passed to it, all the

commands have to receive those arguments. They don't have to use them, just receive them. As you look for all the changes in the program listings, you will notice that only one command uses the arguments passed to it. The remainder of the commands are almost exactly the same as the one shown above.

I encourage you to go over the source code closely to see how it has changed to fully implement the ARexx interface. The program can still be expanded with additional commands, but the interface would still be the same.

Program Comments

The Filer program is being used as a tool to show how you can add an ARexx interface to your own program. It is not very easy to use. I realize that in its current configuration that you may not be too impressed with it. I am also not too impressed with it. However, I am continuing to work with it to improve the interface and functionality of the program. I have some ideas for it that I hope to be able to share with you in future articles to show you how to use other aspects of the Amiga operating system. In the first part of this article, I did not give any instructions on how to use the Filer program. I apologize for that and include them now.

Program Instructions

The program runs in the background to be quickly available to access mini-database files. I originally envisioned these files as files I would create to act like small on-line indexes. As an example, I would like to be able to index all the articles I read in the various magazines I receive. Many times these articles contain items of interest to me now or that may interest me in the future. With this program, I could easily search my database and find out which article in which magazine I need to review. However, you could use it for any quick access items you want.

To activate it, press the Alt-Ctrl-I combination, and the program will activate itself. If the little Filer window is in the upper left-hand corner of your screen, then you don't need to press this key combination. You only need to activate the window to have access to its menu.

Clicking on the closewindow gadget does only that. It closes the window and puts the program to sleep but does not terminate the program. The Alt-Ctrl-f key combination brings it back to life. The only way to terminate the program is through the menu Quit item.

The New item in the Project menu allows you to create a new database file. It brings up a window with four gadgets in the top row.

The first two gadgets indicate the type of field you are defining. The Key field type is used when searching the database and can hold more than one key word as long as they are separated by a space. The Raw field type holds raw information only and can be in any format you like. You can only select one of these gadgets.

The next gadget is the size gadget. You specify the length of this field using this gadget. Because the value entered here is never tested, be careful not to enter a length longer than one screen line. Press the return key after entering your value or the program will never learn about it. I capture only GADGETUP events. Pressing the return key in a string gadget causes a GADGETUP event to be sent.

The last gadget is the name gadget. You specify the field name using this gadget. This is not the content of the field, just its name. Again, press the return key after entering the name.

After you have entered all three values, the field name, size, and type will be printed below the gadgets. You can keep defining fields up to a maximum of 12 fields or click on the closewindow gadget if you define less than the maximum.

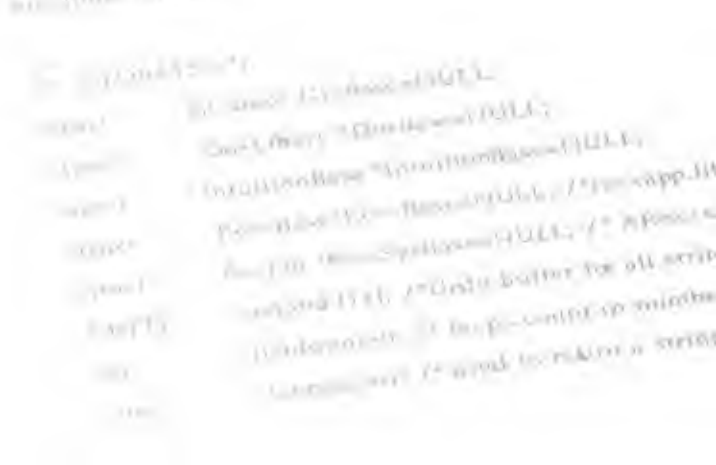
The file definition you have just defined is only in memory and needs to be saved to disk if you want to be able to use it to hold data. To do this, you use the save menu item in the project menu. This prompts you for a filename and then saves the record definition.

The open menu item in the project menu can be used to access a file previously saved. The last opened file remains the current file for all other file activity until a new file is defined or another file is opened.

The add menu item in the edit menu, the new internal command, allows you to add items to your database file. The command opens it own window and presents the field names and string gadgets for each to get your input. Make an entry in each field even if it is just a single character. Use the Add Another gadget in the lower right-hand corner of this window to add as many additional records as you destre. Click on the closewindow gadget when you are done. Again, press the return key in each string gadget.

The last command is the view item in the edit menu, the ARexx macro added as a disk-based command, and is used to search the database file by key field and display all matches. This macro is fairly straightforward and easy to understand.

As I said earlier, this program still requires much work to become a fully functional program, and I intend to continue working on it. If you see potential in this type of utility program, I would appreciate your input and help. You can contact me on GEnie using the e-mail address, D.Blackwell1.



Listing One

```
/* Filer.h
* Header file for the Filer program.  This file should be pre-
compiled to
* save compile time.
 */

/* .... INCLUDES .... */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <functions.h>
#include <exec/memory.h>
#include <exec/execbase.h>
#include <libraries/dosextens.h>
#include <devices/input.h>
#include <devices/inputevent.h>
#include <rexx/rxslib.h>
#include <rexx/rexxio.h>
#include <rexx/errors.h>
#include <intuition/intuitionbase.h>

/* .... DEFINES .... */
#define DOS_REV 33
#define GRAPHICS_REV 33
#define INTUITION_REV 33
#define REXX_REV 01
#define REXXAPP_REV 01
#define BOOL 0
#define FLOC 1
#define NAN 1101
#define KEY 1102
#define SIZE 1103
#define NAME 1104
#define PROJECTMENU 0
#define NEWFILE 0
#define OPENFILE 1
#define SAVEFILE 2
#define QUIT 3
#define EDITMENU 1
#define ADDNEWRECORD 0
#define VIEWRECORD 1
#define ANOTHER 500
#define NUMCMDS 13
#define MAXFIELDS 12
#define REXXAPPNAME "rexxapp library"
#define XORIGIN 0
#define YORIGIN 10
#define XLENGTH 80
#define YLENGTH 10
#define XDELTA  315
#define YDELTA  95
#define PWINDOWFLAGS
CLOSEWINDOW|WINDOWSIZING|ACTIVEWINDOW|INACTIVEWINDOW
#define PWINDOWFLAGS WINDOWCLOSE|ACTIVATE|NOCAREREFRESH
#define NPWINDOWIDCMP CLOSEWINDOW|GADGETUP
#define NPWINDOWFLAGS
WINDOWCLOSE|ACTIVATE|NOCAREREFRESH|WINDOWDEPTH|WINDOWDRAG
#define NRWINDOWIDCMP NPWINDOWIDCMP
#define NRWINDOWFLAGS NPWINDOWFLAGS
/* .... STRUCTURE DECLARATIONS .... */
struct CmdEntry {
                char *CmdString;
                APTR CmdHandler;
        };

struct RexxData {
                struct MsgPort RexxPort;
                char *Extn;
                APTR Func;
                struct RexxMsg *RexxMsg;
                APTR Error;
                APTR Result;
                ULONG RcodMask;
                ULONG UserData;
                struct CmdEntry RexxCmd[NUMCMDS];
        };
struct Definition {
                struct Node defNode;
                UBYTE type;
                UBYTE size;
                char Name[15];
        };
struct EvalCheck {
                char    *success_str;
                char    *error_str;
                char    *return_str;
                long    return_str_length;
                UBYTE   flagbyte;
                ULONG   PortSigMsk;
                ULONG   ReadSigMsk;
                ULONG   AllSigMsk;
                ULONG   waitsigs;
                struct  Window *mywindow;
                struct  Gadget *defGad;
                char    filename[112];
                int     fieldcount;
                BOOL    sleeping;
                BOOL    window_inactive;
                BOOL    cgi_done;
        };
/* .... FUNCTION PROTOTYPES .... */
void Close_Libraries(void);
void FreeDefList(void);
void inforeststruct Window *);
void DeleteRexxMsg(struct RexxMsg *);
void FreeRexxPort(struct RexxData *);
void ParseFunction(struct RexxData *);
void ReplaceGadgetText(struct Gadget *, UBYTE);
void error_rtn(UBYTE, char *, struct RexxMsg *, struct RexxData
*);
void process_rtn(UBYTE, char *, struct RexxMsg *, struct
RexxData *);
void setupRexxMsg(struct USER, LONG, STRPTR, struct RexxMsg *, struct
RexxData *);
BOOL closelib(void);
BOOL rexxclosert(struct RexxMsg *, char *, struct RexxData *);
BOOL makequit(void);
BOOL rexxmakequit(struct RexxMsg *, char *, struct RexxData *);
BOOL activatelib(void);
BOOL rexxactivate(struct RexxMsg *, char *, struct RexxData *);
BOOL newfile(void);
BOOL rexxnewfile(struct RexxMsg *, char *, struct RexxData *);
BOOL openfile(void);
BOOL rexxopenfile(struct RexxMsg *, char *, struct RexxData *);
BOOL savefile(void);
BOOL rexxsavefile(struct RexxMsg *, char *, struct RexxData *);
BOOL quitprog(void);
BOOL rexxquitprog(struct RexxMsg *, char *, struct RexxData *);
BOOL rexxsupplyfilename(struct RexxMsg *, char *, struct
RexxData *);
BOOL rexxqueryrecordlist(struct RexxMsg *, char *, struct
RexxData *);
BOOL rexxsupplyrecordlist(struct RexxMsg *, char *, struct
RexxData *);
BOOL viewrecord(void);
BOOL rexxviewrecord(struct RexxMsg *, char *, struct RexxData
*);
BOOL addnewrecord(void);
BOOL rexxaddnewrecord(struct RexxMsg *, char *, struct RexxData
*);
BOOL writerecord(struct Gadget *, struct Window *);
BOOL Open_Libraries(void);
```

Listing Two

```
   Parse Var number 'Name',index Type,index size,index length
   If Type=value <> 0 then iterate /* skip if not a key field
*/
   KeyName.keycount = index
   Keycount = keycount + 1
end

If keycount < 1 Then Exit 5 /* Can't install a file w/o a key
field */

  recordsize' /* Request the record size  */

If RC <> 0 Then Exit RC
Size = result
If Size = '' Then Exit 1  /* Exit with a warning. */

recordsize = Value(Filel /* Convert file argument value to a
numeric value. */

If ~ Open('SetFile',Current_Name, 'R') Then Exit 1

/* Open an input channel */
Ctl_Socket = Open('KeyConsole' , 'Con:////////Keystring')
Ctl_socket = writech('KeyConsole', '1');
Ctl_Socket = writech('KeyConsole','Enter search key: ');
search_string = Readln('KeyConsole') /* Get the key to search
for.  */
Ctl_Socket = Close('KeyConsole')

If search_string = '' Then DO; /* Can't continue w/o a
search string. */

/* Skip the record definition */
Ctl_Socket = Readch('Datafile', (8 * Total_Fields) + 4)

Do While ~ EOF('Datafile')
   Current_Record = Readch('Datafile', Record_Size)
   record1 = Current_Record /* Parse using a temporary
variable. */

   /* The record fields are stored in reverse order and need to
be.*/
   /* Output in the same way  */
   Index = Total_Fields + 1
   Do Loop_Count = 0 to Total_Fields - 1
      position = Loop.index * 1
      Parse VAR temp1 1 field.index +position temp1
      Ctl_Socket = Writech(field,index)
      index = index - 1
   End

   Do index = 0 to Keycount - 1
      Key = Keyname.index
      Total_Keys = Value(Field.key)
      Temp1 = Field.key
      Do index2 = 1 to Total_Keys - 1
         Parse VAR temp1 temp2 temp1
         If Search_String = temp2 Then Found_Record=
      End
   End
end

Ctl_Socket = Close('Datafile')

Exit

Show:
Arg record
address = 0

Do record = 0 to Total_Fields - 1
   If Size record = longest Then Longest = Size.index1
End

Length = (Longest + 1) * 2
Width = (Total_Fields + 1) * 10
```

```
Xcenter = (ZOO - Height)/2
Xcenter = (640 - Length)/2

bit_bucket = Open('ShowRecord', 'CON:'Xcenter'/'Ycenter'/
'length'/'height'/Show Record')

Do index1 = 0 to Total_Fields - 1
   bit_bucket = Writeln('ShowRecord', Node.index+1 ||
Field.index1)
End

bit_bucket = WriteCh('ShowRecord', 'Press any key to con
tinue...')
bit_bucket = ReadCh('ShowRecord', 1)
return 0
```

Listing Three

```
/* Filer.c */
/* Manx 5.00 source */
/* cc -mc -m0 -aa -s0 -s0, -s0 -s0 -f0 Filer.c */
/* ln Filer.o ReqXInterface.o c.lib */

#include "Filer.h"

/* ... DiskObj ... */
struct   DiskBase *DiskBase = NULL;
struct   IconLibrary *IconBase = NULL;
struct   IntuitionBase *IntuitionBase = NULL;
struct   ReqXBase *ReqXBase = NULL; /* reqxapp.library pointer
*/
struct   ReqXLib *ReqXSysBase = NULL; /* AReqX system library */
UBYTE    (*UndoBuffer)(); /* Undo buffer for all string gadgets */
ULONG    EventSignal; /* wake-up signal for event handler */
struct   Interrupt *myHandler;
struct   IOStdReq *myIOreqest;
struct   MsgPort *myPort;
struct  MsgPort *ASyncPort;
struct   Task *myTask;      /* pointer to Filer task structure
*/
struct  myGlobals *Global; /* a structure to hold global
variables */
extern  char *ReqXErrMsg; /* error message set by
reqxapp.library */

/* Comments on some fields of the myGlobals structure:
 FileCount       - a count of the definition structures in a
record
 defList         - linked list of field definitions that
comprise a record
 return_str      - pointer used to pass return strings back to
AReqX macros
 FileOpen        - the active file handle
 sleeping        - These next three variables describe the
current
 window_inactive - state of the Filer window throughout the
 All_done        - execution of the program
 wakeSig         - Signals that could wake-up the program
*/

struct ReqXData myRD = {
                         (NULL),
                         (char *) "Fil",
                         (APTR) dispatch,
                         (0L),
                         (APTR) error_rtn,
                         (APTR) process_rtn,
                         (NULL),
                         (NULL),
                         { { "Now",
    (APTR)&reqxxNewFile },
                                                  { "Open",
    (APTR)&reqxxOpenFile },
                                                  { "Save",
    (APTR)&reqxxSaveFile },
                                                  { "Quit",
    (APTR)&reqxxQuitReq },
                                                  { "Archive",
    (APTR)&reqxxArchive },
                                                  { "Weekend",
    (APTR)&reqxxWeekend },
                                                  { "Cleanup",
    (APTR)&reqxxCleanup },
                                                  { "FileDone",
    (APTR)&reqxxApplyFileDone },
                                                  { "ScrollFile",
(APTR)&reqxxApplyScrollFile },
                                                  { "RemoveFile",
(APTR)&reqxxApplyRecordFile },
                                                  { "View",
    (APTR)&reqxxViewRecord },
                                                  { "Add",
    (APTR)&reqxxAddNewRecord },
                                                  { (UBA) 0 }, (APTR) 0 } }
                    };

struct TextAttr MyFont = {(STRPTR)"topaz.font",TOPAZ_EIGHTY,
        FS_NORMAL,FPF_ROMFONT};

/* Include all the Menu, Gadget, Requester and miscellaneous
definitions */
#include "filer.gw" /* Modified ReverseWindows output file */

void main (void)
{
   LONG myClass;
   WORD myCode;
   ULONG etemp[4];
   struct NewMap *myNewMap;
   struct IntuiMessage *myMessage;
   static char PortName[8] = "APort"; /* my AReqX port name */

   if (!OpenLibs())
      goto main_exit; /* OpenLibs() failed, can't continue
*/

   if (!(myPort = CreateMsgPort(PortName, NULL)))
   {  /* a message port is critical to the program's operation
*/
      puts("Unable to create a message port.  Exiting!");
      goto main_exit;
   }

   if (!(ASyncPort = CreateMsgPort(ASyncPort, (long) 0)))
   {
      puts("Unable to create an asynchronous message port.");
      goto main_exit;
   }
   if (!(Global = AllocMem((sizeof(struct myGlobals),
MEMF_CLEAR)))
   {  /* Unable to allocate needed memory for global variables
*/
      puts("Not enough memory for global variables.");
      goto main_exit;
   }
   Global->ArrowKey = FALSE;
   Global->window_inactive = FALSE;
   Global->all_done = FALSE; /* Execution of the program */

   if (!(myWin = OpenWindow( &Open_Window[0], XORIGIN,
(short) YORIGIN,
         (long) XLENGTH, (short) YLENGTH, NULL, (WFLG)WFLAGS,
         (UBYTE *)"Filer", (struct Screen *)0)))
   {
      puts("Insufficient memory for the Filer Window.
Exiting!");
      goto main_exit;
   }
```

```
    ((rxObj)->myRxPort->UserPort = myPort; /* Add the rexxmsg port
tion modify */
    ModifyIDCMP((Global->myWindow), (WINDOWIDCMP)); /* the window's
IDCMP flags */

    if (RexxBase) /* if RexxBase is not 0 exec.lib library is
available */
    {
        if (!(Global->aRexxSignal =
SetRexxPort((CONST_STRPTR)FindName(...&myRX)))
        {
            Global->error_str = "Can't open Rexx message
port */
            CloseWindow(Global->myWindow);
            CloseLibrary((struct Library *) RexxBase); /* Can't
open a port */
            RexxBase = (struct RexxBase *)0; /* Or close the
library */
        }
    }

    myTask = FindTask(0); /* The next two globals are intended
solely for */
    EventSignal = 1L << AllocSignal(-1L); /* used by my event
handler */
    Global->PortSignal = 1L << myPort->mp_SigBit;
    Global->ASyncSignal = 1L << ASyncPort->mp_SigBit;

    /* This assignment works correctly regardless of
newSignal's value */
    Global->waitmask = (Global->PortSignal|EventSignal|Global->
RexxDismal|Global->ASyncSignal);

    void(*)()main_handler(myPort); /* start my input handler */

    SetMenuStrip(&(Global->myWindow), MenuList);

    while (!Global->all_done)
    {
        sigmsgs = Wait(Global->waitmask);
        if (sigmsgs & EventSignal)
        { /* if the program is sleeping or the window is
inactive and is */
            /* receives an input event signal, then make up is
activate the */
            /* window and process the user's request */
            if (Global->window_inactive)
                activate();
            if (Global->sleeping)
                wakeup();
        }
        if (!(Global->sleeping) && (sigmsgs & Global->
PortSignal))
        { /* Process all IDCMP messages that arrive at my
window's message */
            /* port. */
            while (myMessage = (struct IntuiMessage
*)GetMsg(myPort))
            {
                myClass = myMessage->Class;
                myCode = myMessage->Code;
                ReplyMsg((struct Message *)myMessage);
                switch(myClass)
                {
                    case CLOSEWINDOW:
                        close();
                        break;
                    case MENUPICK:
                        Global->all_done =
processMenu(myCode);
                        break;
                    case ACTIVEWINDOW:
                        Global->sleeping = FALSE;
                        Global->window_inactive = FALSE;
                        break;
                    case INACTIVEWINDOW:
                        Global->sleeping = FALSE;
                        Global->window_inactive = TRUE;
                        break;
                    default:
                        Global->error_str = "Unexpected
class of IntuiMessage";
                        InformGlobal->myWindow);
                }
            }
        }
        if (RexxBase && (sigmsgs & Global->RexxSignal))
        { /* If ARexx is available and we receive a signal
from our ARexx */
            /* message port, call the rexxsupp.library
ReceiveRexxMsg() function */
            /* to process all the outstanding ARexx messages. So
simple! */
            ReceiveRexxMsg(myRX);
        }
        if (sigmsgs & Global->ASyncSignal)
        {
            while (myRexxMsg = (struct RexxMsg *)
GetMsg(ASyncPort))
            {
                if (myRexxMsg->rm_Args[0])
                    DeleteArgstring(myRexxMsg->rm_Args[0]);
                DeleteRexxMsg(myRexxMsg);
            }
        }
    }

    close_input_handler(); /* shut down input handler */

main_exit:
    if (Global->myWindow)
    {
        if (Global->myWindow->MenuStrip)
            ClearMenuStrip(Global->myWindow);
            Global->myWindow->UserPort = NULL;
            CloseWindow(Global->myWindow);
    }

    if (downchort)
    {
        while (myRexxMsg = (struct RexxMsg *) GetMsg(ASyncPort))
        {
            if (myRexxMsg->rm_Args[0])
                DeleteArgstring(myRexxMsg->rm_Args[0]);
            DeleteRexxMsg(myRexxMsg);
        }
        DeletePort(ASyncPort);
    }

    if (myPort)
    { /* reply to all outstanding messages and delete the
message port */
        while (myMessage = (struct IntuiMessage *)GetMsg(myPort))
            ReplyMsg((struct Message *)myMessage);
        DeletePort(myPort);
    }

    if (Global->flags)
        Close(Global->flags);

    if (Global->antlist)
        FreeAntList();

    if (Global->RexxSignal)
        FreeSignal(myRX);

    if (Global->return_str)
        FreeMem((Global->return_str, Global->return_str_length);

    close_libs();

    return;
}
```

The remaining code for Listing Three can be found on the **AC's TECH Disk** along with all other files necessary for this program.

Please write to:
Dave Blackwell
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Iterated Function Systems For Amiga Computer Graphics

by Laura M. Morrison

Traditional graphics interprets objects in terms of geometric figures such as squares, circles, triangles, and polygons. Children are taught to draw by piling up spheres and cubes. Many natural objects, however, do not lend themselves to this approach. They have the property of "self-similarity" and are more easily and realistically rendered in terms of smaller versions of themselves. M.F. Barnsley and A.D. Sloan used iterated function systems (IFS) to encode images of natural objects in terms of smaller versions. By exploiting the property of self-similarity inherent to natural objects, they encode intricate natural images using only the coefficients Affine transformations. Although the number of transformations necessary to encode a picture ranges from a few to hundreds, IFS allow picture storage compression with ratios of the order of 10,000 to 1 (Reference 1).

Besides their usefulness for image-storage compression, iterated function systems can serve the computer graphics artist. Even a computer artist who is unable to draw and knows no mathematics can use iterated function systems to render convincingly natural images of objects such as trees, plants, clouds, mountains and flowers. See Examples 1 and 2.

The difficulty has been finding the Affine transformations which map the whole image into "tiles," or smaller versions, that cover the whole. These transformations have the form:

$$x2 = ax1 + by1 + e$$
$$y2 = cx1 + dy1 + f$$

where (x1, y1) is a point on the whole and (x2, y2) is a point on the tile. They can rotate, skew, translate, or resize. Tiles, therefore, can be a versitile elements for rendering of the whole. The most important characteristic of a graphically useful Affine transformation is that it maps the larger whole into itself. A tile must be completely inside the whole. One such transformation, if iteratively applied to a medium of infinite resolution, would map the whole into a tile, then tile into sub-tile, then sub-tile into sub-sub-tile, and so on—ad infinitum. If, however, many such transformations are iterated randomly, the result is an intricate, natural-looking image or an intricate design (Illustration 2, and Examples 1, 2, and 3).

Finding these transformations has required trial and error, and serendipity for the personal computer graphics artist. A program that works for regular n-gons is available for the Amiga (Reference 2). Trial-and-error algorithms are available for MS-DOS computers (References 3 and 4).

The algorithm described here uses mouse-click input to define the transformations. Although this computer program is for the Amiga and relies on many Amiga features, the algorithm is portable. The section of coding that actually computes the coefficients is not Amiga specific.

The program reads and displays a user-prepared sketch of the whole image showing indications of tile placement. See, for example, Illustration 1. The purpose of this sketch is to guide the user's mouse clicks. The user-artist clicks the left mouse button one tile at a time, point on the whole to analogous point on the tile, to tell the program how the whole image should be mapped into its many covering tiles. The program records the mouse click positions and provides feedback of what it thinks the user is doing. When three pairs of points have been collected, the program uses Cramer's rule to solve six simultaneous equations for the six coefficients, a, b, c, d, e, and f, of the transformation. A general purpose simultaneous equations solver is unnecessary since never more than three equations in three unknows need be solved at once. The six points input by the user provide three equations in the unknowns a , b, and e:

$$x2 = ax1 + by1 + e$$

and three equations in the unknowns c, d, and f:

Table One

```
These are the coefficients produced during a (random) walk
through using the mandelbrot guide equation (z = z^2 + c).
Each row contains the six coefficients a, b, c, d, e, f, for
the transformation of points on the whole to successive tiles:
      (x1,y1) ---->  (ax1+by1+c, dx1+ey1+f)  =  (x2,y2)
The second from last line marks the end of transformations.
The last line contains escape, extent, y-scale and
yoffset parameters which adjust the computed values to fit
plot on the amiga screen.
```

a	b	c	d	e	f
-0.04107	-0.00126	0.60911	0.01001	181	-201
0.37424	-0.00094	0.04817	0.18000	90	-113
0.03679	-0.24729	0.21181	0.18493	91	90
0.30848	0.24749	0.01201	0.33496	121	110
-0.66428	-0.26007	-0.06894	0.49021	180	43
0.54136	0.26948	0.07749	0.41105	230	71
999.90005	999.90005	999.90005	999.90005	1000	1000
	1.000000	0	1.000000	100	

$$y2 = cx1 + dy1 + f$$

IFS_coder rejects transformations with determinant ac-bd = 0. Other than that, does no error checking. If this were an engineering application, error analysts and control would be essential but for this graphics application checking is not even useful. If the user-input-points do not correspond to mapping analogous points from the whole to a tile, then aberrant transformations can result but such transformations have not been excluded because they are easily spotted and deleted from the coefficient file and wild transformations can produce artistically pleasing abstractions at the computer artist's discretion, as well as possibly provide bizarre fission-type animations for galactic game programmers. Try Table 5 Fission coefficient file as input to IFS_decoder

Table Two

```
The coefficients for Tree (Seed, See Example 1. These can be
input to IFS_decoder to produce the Tree Image.
```

a	b	c	d	e	f
0.02496	0.01539	-0.28124	0.01687	261	66
0.21838	-0.20984	0.24265	0.18766	241	148
0.14646	0.39802	0.08838	0.30666	83	160
0.20519	0.29528	0.12084	0.21218	242	140
0.11181	0.16749	0.22348	0.19329	251	197
-0.00304	0.27517	0.19618	-0.01290	180	226
0.15987	0.29932	0.20748	-0.00588	203	111
0.29200	-0.18719	0.20984	0.20770	201	134
-0.20786	0.28598	0.19892	0.03161	167	183
0.01279	-0.00624	0.23111	0.24449	189	9
999.90005	999.90005	999.90005	999.90005	1000	1000
	1.000000	0	-1.000000	100	

IFS_coder writes a transformation's coefficients as one row to an ASCII file. See, for examples, Tables 1,2,3 and 4, coefficients for a Weed, a Tree, an Orchid, and a Square. This file should be ASCII because the user will want to edit it, adding transformations, removing erroneous transformations, or just "polishing" coefficients.

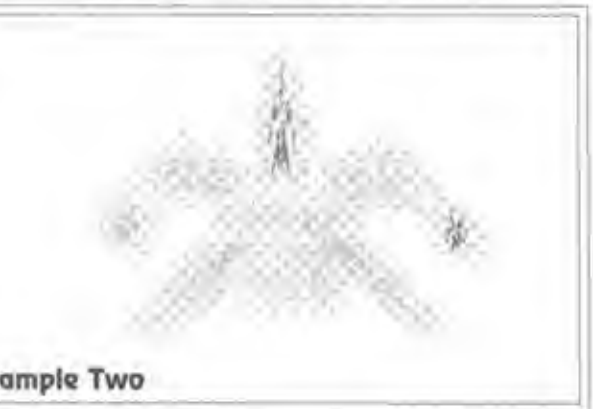
The user quits encoding whenever he wishes by clicking a "CloseWindow" gadget hidden in the upper left corner of the screen. The program will do numerous transformations or merely add one transformation to an old coefficient file. With some extra coding it could remove the old "end of file" lines. As it is, the user must delete these before submitting the revised coefficient file to IFS_decoder.

Example One

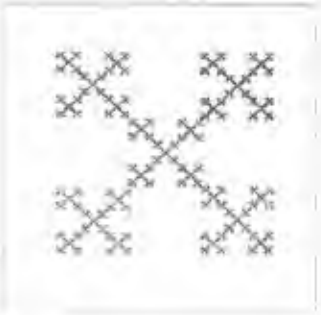
The IFS_decoder program reads the coefficient file and iterates the transformations randomly to produce a completed image (Listing 2 IFS_decoder). There is nothing necessary or magical about picking the next transformation randomly. It is merely the most practical way of applying the transformations in all possible permutations of order.

The literature suggests that each transformation be assigned a probability of being iterated commensurate with the area its tile covers. This would unnecessarily complicate the iteration code and doesn't change the final image. Should the user wish to increase a transformation's weight, he can duplicate or triplicate its row in the coefficient file and get the same effect.

The program uses color to identify a tile's transformation. The register number of a tile's color in the final image corresponds (modulo 15) to the row number of its transformation in



Example Two



Example Three

the coefficient file. The SetRGB4(rastport,register,%red, %green,%blue) statements show the correlation between a color and its register. The computer artist will, of course, want to take the final image to a paint program and alter its colors.

Iteration continues until the user is satisfied with the image's density and stops the iteration by clicking the CloseWindow gadget. The program automatically writes an image file. Or if the user is disatisfied with the image, he may abort the program by keying 'a' to quit without saving the image.

Reading and writing image files is prehaps the most Amiga-idocyncratic part of the coding. Since the Amiga is multitasking, a program can run other programs from within itself. The

"Execute(<command_line>,0,0)"

statements do that. They run Commodore-supplied public domain program "raw2ilbm" (not listed) to translate images between the raw bitmap image used by IFS_coder and IFS_decoder, and the IFF standard image used in Paint Programs. Notice, however, that the guide-sketch is not converted by IFS-coder. It will be used repeatedly and conversion takes too long. The guide-sketch must be available raw in the same directory as the program. The final raw image file is

Illustration Two

large so the program writes it to RAM:, converts it to a standard IFF picture file and then deletes it. This ability to run programs within programs makes prototyping an application faster and easier. To run the converter programs from the Amiga command line, type:

ilbm2raw <imagefilename>
raw2ilbm <imagefilename> <newILBMfilename> hi 4

The number of transformations the IFS_decoder can iterate at one time needs to be unlimited. Notice that all the transformations must be iterated at one time since they interact with one another. It is not possible to, say, iterate half and then add an over image produced by the other half. Although a List, with dynamic dimensioning, to hold the transformation coefficients would allow unlimited transformations, Arrays,

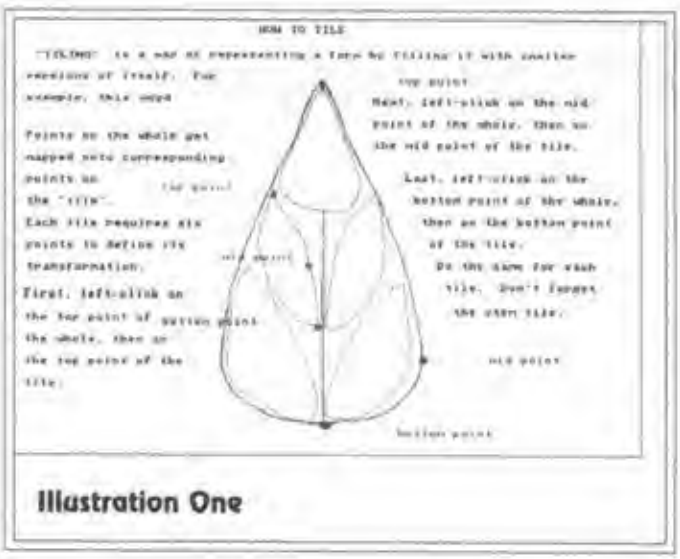


Illustration One

even though of fixed dimensions, are better. Coding for iterations with Arrays is simplier and faster. Change MAXTRANS to change the Array dimensions to iterate more than 32 transformations.

The IFS_coder program saves a list of the selected points to an ASCII file, called "<coefficent_filename>points". This file contains a list of the pairs of points selected by the user. There are six pairs of points for each transformation. The points from a walk-through of WeedTutor are given in Listing 5. This list can be ignored unless the user wishes to revise or polish the points. The new points can be input to the Points2Coefficients program to produce a revised coefficient file (Listing 3).

The IFS_decoder program produces a list, called "points," of the first 50 points it generates. See, for example, Listing 6 which gives the first 50 points of the Weed image shown in Illustration 2. This list is necessary in case the decoded points do not fall on the display screen. The Amiga display, with its origin in the upper left corner and its growth downwards,

Table Three

List of coefficients for the second image.

a	b	c	d	e	f
0.01281	0.00164	-0.00049	0.01704	271	208
0.42931	0.42668	0.15002	0.18681	195	184
0.14095	0.11110	0.18916	0.13937	413	48
0.04990	0.01829	-0.01811	-0.48811	280	247
0.09274	0.00199	-0.00715	-0.44105	59	81
0.25041	-0.04054	0.01800	0.23954	139	208
0.2725%	0.09028	0.01604	0.21810	301	201
0.06760	0.06263	-0.04159	0.47743	193	108
0.09481	-0.03319	-0.01428	0.26240	467	151
999.90005	999.90005	999.90005	999.90005	1000	1000
	1.00000	-1.00000	999		

Table Four

List of coefficients for the square image
See Example 2.

a	b	c	d	e	f
0.37535	-0.00141	-0.00959	0.49629	131	141
0.01533	0.00000	0.01553	0.33181	320	240
0.38202	0.00436	-0.00280	0.31750	208	152
0.34191	0.00000	0.00000	0.37340	141	76
0.32068	-0.00367	-0.00321	0.34182	289	70
999.90005	999.90005	999.90005	999.90005	1000	1000
	1.00000	-1.00000	999		

Table Five

List of coefficients for Fractal animation.

a	b	c	d	e	f
-0.5219	-0.0149	-1.3971	0.5997	454	255
0.8966	0.0796	-0.2290	0.7824	311	59
-0.1509	0.4761	-0.9819	-0.0511	-10	500
999.9	999.9	999.9	999.9	999.9	999.9
	0.01	180	0.21	170	

differs from the standard mathematics first quadrant. The IFS_coder program automatically takes care of this for most transformations it defines. If not, the list shows where the points are going, and the user can adjust the scale and offset values at the end of the coefficient file so the points will fall within the display screen.

The trick to tiling is to make a separate transformation for each tile and put each tile completely within the whole. Each tile requires three pairs of mouse clicks: a mouse click on the whole, then a mouse click on the analogous tile-point, three times. Illustration 1 describes what must be done. The more characteristic the points selected, the more faithfully the tile mirrors the whole.

Indicators tell what the program thinks the user is doing, whether a click has defined a point on the whole ("From" point) or a point on the tile ("To" point). The number of pairs of points already clicked for the current tile is posted. (One transformation requires exactly three pairs of points.) The program also posts the number of transformations already completed. With very little practice, the user will ignore these indicators.

A tutorial script walks the beginner through a tiling (Listing 4 for Tutor). Everything needed is in the tutorial directory. To run the tutorial, enter the Amiga CLI, change directories to the tutorial directory, and type:

Execute Tutor

Look for the resulting image in RAM: where the IFS_decoder program has put it. To see the final image type:

View <ram:filename.pic>

Copy the file to disk to save it. Illustration 2 shows how the resulting image should turn out.

Biographical sketch

Laura M. Morrison has a Master's Degree in Mathematics from New York University, N.Y. She has worked as an Operations Research Analyst, designing applications software for Esso R&E, Union Carbide, and Eastern Airlines. She is at present self-employed.

Listing One

No one covers the

With the premiere monthly Amiga magazine, *Amazing Computing* , the technical depth of *AC's TECH*, and the complete Amiga market insight of *AC's GUIDE*, you have no better choice for keeping you up to date on the rapidly changing Amiga marketplace.

Amazing Computing for the Commodore AMIGA

Amazing Computing specializes in providing its readers with a broad knowledge of Amiga computing. With articles on the latest trade shows, the latest released products, indepth how-to's, plus informative programming and hardware features, *AC* maintains a standard of delivering only the very best.

Whether through the controversial columns of *Roomers* or the always informative *Bug Bytes*, *AC* maintains the constant process of providing alternative views as well as help in making your Amiga do more for you. *AC* does more than any other Amiga publication to provide the latest information and the best news stories on what the Amiga will be doing next. When you want to know more and do more with your Amiga—*AC* is the one choice.

***Amazing Computing* provides its readers the following:**

In-depth reviews and tutorials
Informative columns
Latest announcements as soon as they are released
Worldwide Amiga Trade Show coverage
Programming tips and tutorials
Hardware projects
The latest in non-commercial software
All in informative, but in an easy to understand format

AC's TECH For The Commodore Amiga

AC's TECH was created with the more experienced, or determined, Amiga user in mind. It is the perfect complement to *Amazing Computing* and *AC's GUIDE*. *AC's TECH* attempts to take the mystery out of intense programming and hardware development. The issues covered in *AC's TECH* are of interest to both the beginning programmer and the Amiga developer.

AC's TECH allows Amiga users to expand their knowledge and commit to larger projects while staying aware of the latest releases and other changes in the Amiga platform. This vital resource was the first disk-based Amiga technical resource available. It is, once again, the only such resource available and continues to improve with each issue.

***AC's TECH* offers these great benefits:**

The only disk-based Amiga technical magazine
Hardware projects
Software tutorials
Interesting and insightful techniques and programs
Full listings on disk
Amiga beginner and developer topics

Amiga like Amazing

AC's GUIDE To The Commodore Amiga

AC's GUIDE is recognized as the world's best authority on Amiga products and services. Amiga Dealers swear by this volume as their bible for Amiga information. With complete listings of every software product, hardware product, service, vendor, and even user groups, *AC's GUIDE* is the one source for everything in the Amiga market.

AC's GUIDE also includes a directory of Freely Redistributable Software from the Fred Fish Collection and others. For Commodore executives, Amiga dealers, Amiga developers, and Amiga users everywhere, there is no better reference for the Commodore Amiga than *AC's GUIDE to the Commodore Amiga*.

What do you get with AC's GUIDE?
The best single resource for everything available in the Amiga market!

Subscriber Benefits

Every investment should contain some added value that makes your choice more interesting. An Amazing subscriber not only gets one of the best magazines available, but these additional perks as well!

Every Amazing publication has always been mailed in a protective cover or envelope.
This means the end to torn covers or mutilated issues. We want your AC investment to always remain a solid value.

Every AC Subscriber issue is mailed early.
Every *AC* is mailed one week before the copies are shipped to the newsstands by the distributors.

Toll-Free Access.
An 800 number for toll-free (from the U.S. and Canada) assistance on your subscription. Amazing handles all subscription questions at our corporate offices. This means that your concerns concern us and we take care of your problems FAST!

Money Back Guarantee.
If you find your *Amazing* subscription is not everything you need for a better Amiga, contact us for a hassle-free refund for any unmailed issues.

Now is your chance to enjoy the best savings ever available for these publications.
This Amazing special is only available for a limited time.
Use your MasterCard or Visa and call toll-free in the U.S. and Canada:

1-800-345-3360

HURRY!

Due to the tremendous response received so far, The AMAZING SPECIAL deadline has been extended!

Listing Two

```
Listing 2

/* IFS_Decode.c  Version 2.0  4/15/91  */
#include "stdio.h"
#include "exec/types.h"
#include "exec/tasks.h"
#include "exec/memory.h"
#include "exec/execbase.h"
#include "libraries/dos.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/view.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/types.h"
#include "math.h"

#define COARSEST   12
#define MAXFRAME    10
/* The number of transformations is conveniently set below */
#define DEPTH    5
#define WIDTH    320
#define HEIGHT   400
/* Arbitrary colors to move with the final leaves */
UBYTE cmap[] =
{0x000,0x0f0,0x0bf,0x0fb,0x0ff,0x8f8,0xbff,
                0x8a0,0x090,0x0b9,0x0ff,0x0f0,0x0ff,0x0f0,0x0f0,0x0f0
};
extern struct RxsNdu *Nu;
extern struct IntuiMsg *Fun();
/*  C Structures defining the screen and viewport to be opened
later.
    The resolution is set by Initialize.  The message
system
    requires a window be opened as well as a screen.  The
packages we
```

```
    have to provide are needed under circumstances
where
    they
    */

struct NewScreen newscr =
    {0,0,320,400,4,0,1,
    HIRESLACE,
    CUSTOMSCREEN,
    NULL,
    "IFS_Decoder Screen",
    NULL,NULL,
    };
struct IntuiMsg *im;
struct NewWindow nw =
    {0,0,320,400,0,1,
    MOUSEBUTTONS|CLOSEWINDOW|RAWKEY|VANILLAKEY,
    WINDOWCLOSE|ACTIVATE|NOCAREREFRESH|SIMPLE_REFRESH,
    NULL,NULL,NULL,
    NULL,NULL,
    0,0,320,400
    ,CUSTOMSCREEN };
struct Window *w;
UBYTE *scrn;
extern struct RastPort *rp;
struct ViewModes *Query();
struct ViewPort *Vp;
struct RastPort *rp;
struct View *v;
struct IntuitionBase *IntuitionBase
struct GfxBase *GfxBase;

/*  Arrays for the transformation coefficients.  Arrays
    are better than lists which would allow for variable
    dimensions but complicate and slow the iterations later.
*/

double a[MAXFRAME];
double b[MAXFRAME];
double c[MAXFRAME];
double d[MAXFRAME];
double e[MAXFRAME];
double f[MAXFRAME];
double x0, y0;
double x1, y1;

main(argc,argv)
char **argv;
{
    int argc;
    char *argv[];
{
        LONG rda;
        struct IntuiMessage *msg;
        int  quitflag;
        ULONG class;
        USHORT code;
        SHORT  x, y;
        int xoffset, yoffset;
        LONG ramta, ramfp, i;
        int  mode, ch, r, rtemp;
        FILE  *pcode,*ptable;
        char *fopen(),*fgets();
        LONG IFS_Decode();
        double sqrt();
/*  Compile get_IFS & IFS_Init_Image.  */

    IntuitionBase = (struct IntuitionBase *)
                    OpenLibrary("intuition.library",0);
    if((IntuitionBase == NULL)
    {  printf("Couldn't open Intuition library\n");
       exit( aborted);  }

    GfxBase = (struct GfxBase
*)OpenLibrary("graphics.library",0);
    if(GfxBase == NULL)
    {  printf("Couldn't open Graphics library\n");
```

```
        goto aborted; }

   printf("\n\n      IFS_DECODER Program Copyright 1991 \n");
   printf("                by Louis M. Morrison  \n");
   printf("                   80 North Moore Street, Apt 4J\n");
   printf("                   New York, N. Y. 10013  \n");
   printf("                   Telephone: 212/349-1918  \n");

   if (argc < 2 )
   { printf(" Need coefficient filename on the command
line.\n");
     goto aborted; }

   if ( argc < 3 )
   {
     printf(" Need output image filename  on command line.
\n");
     printf(" None given so using a default name. \n");
   }

   if (( pp = fopen(argv[1],"r")) == NULL)
   { printf("Trouble opening the coefficient file.\n");
     goto aborted;
   }
  i=0;
readmore:;

  fscanf(pp," %f %f %f %f %f %f
",&a[i],&b[i],&c[i],&d[i],&e[i],&f[i]);
   if( a[i] < 999.0   )
   {
       i++;
       if( i > (MAXTRANS-1) )
       {
          printf("Too many transformations for allocated
storage.\n");
          goto abort99;
       }
       goto readmore;
   }
   ntrans = (i-1);
  fscanf(pp," %f  %d  %f  %d
\n",&xscale,&xoffset,&yscale,&yoffset);
  fclose(pp);
  bitm = (struct BitMap *)AllocMem(sizeof(struct
BitMap),MEMF_CHIP);
       if (bitm==NULL)
       {  printf("Couldn't allocate memory for bmap structure
\n");
          goto aborted; }
      InitBitMap(bitm,DEPTH,WIDTH,HEIGHT);
      planesize = (LONG)(WIDTH/8)*HEIGHT;
      for (i=0; i < DEPTH; i++)
      { bitm->Planes[i]=(PLANEPTR)AllocRaster(WIDTH,HEIGHT);
          if (bitm->Planes[i] == NULL)
          {  printf("bitmap planes were null\n");
             goto aborted; }
        BltClear(bitm->Planes[i],planesize,1);   }
      scr = (struct Screen *)OpenScreen(&newscr);
        if (scr==NULL)
        { printf("scr was null\n");
          goto aborted; }
      nw.Screen = scr;
      nw.BitMap = bitm;
      w = (OpenWindow(&nw);
        if (w==NULL)   goto aborted;
      rp = w->RPort;
      vp = &w->WScreen->ViewPort;
/* Tiles are plotted different colors according to the
   transformations that define them. (modulo 15)  The SetRGB4()
   statements tell which register a color is in.
   SetRGB4(viewport,register number  &red , &green, &blue )  */

   SetRGB4(vp,0,0,0,0);
   SetRGB4(vp,1,15,0,0);
   SetRGB4(vp,2,0,0,15);
   SetRGB4(vp,3,0,15,0);
   SetRGB4(vp,4,15,15,0);
   SetRGB4(vp,5,0,15,15);
   SetRGB4(vp,6,15,0,15);
   SetRGB4(vp,7,5,9,11);
   SetRGB4(vp,8,8,4,5);
   SetRGB4(vp,9,9,2,0);
   SetRGB4(vp,10,0,4,8);
   SetRGB4(vp,11,0,9,5);
   SetRGB4(vp,12,6,8,2);
   SetRGB4(vp,13,3,5,10);
   SetRGB4(vp,14,7,12,7);
   SetRGB4(vp,15,13,5,6);

   SetRast(rp,0);

   x1 = 0;
   y1 = 0;
   count=0;
/* Open a file called "points" to contain the first fifty
points
   generated by the iteration.  In case no image appears on the
   screen check these points to see where they are going.
   Adjust scale and offset parameters so points plot on the
display.*/

  tp = fopen("ram:points","w");
for (;;)
{   while( msg=(struct IntuiMessage *)GetMsg(w->UserPort))!=
NULL)
    {
              class = msg->Class;
              code  = msg->Code;
              x = msg->MouseX;
              y = msg->MouseY;
              ReplyMsg(msg);

/* The program continues to plot points until stopped.
   When the image has a satisfactory density click the
   upper left corner where the CloseWindow gadget is hidden.
   Closing takes a long time because the program writes a raw
   file then converts it to an IBM file.  It works in  ram
   rather than to disk because the raw file can be quite large.
   Copy the image file to disk to save it. */

              if (class == CLOSEWINDOW)  goto finish;
/*  Key 'e' to abort the program without saving the image. */

              if ((class== VANILLAKEY)&&(code=='e'))   goto
aborted;
    }
/* By processing transformations randomly the program
   eventually gets all sequences of transformations and
   thereby all the points.  There is nothing magical or
   or necessary about random selection.  It is just the
   easiest way to exhaust the possibilities. */
          i = rand()  % ntrans;
/* Given the last point(x1,y1) calculate and plot the next
   point(x2,y2) */
          x2 = a[i]*x1   +  b[i]*y1  +  c[i];
          y2 = d[i]*x1   +  e[i]*y1  +  f[i];
                    x1 = x2;
                    y1 = y2;
    /* Calculate in double floating point, plot in integers */
                  nextx = (LONG)(xscale*x2 + xoffset);
                  nexty = (LONG)(yscale*y2 + yoffset);
                    cc = (i  % 14)  +  1;
                    SetAPen(rp,cc);
                    WritePixel(rp,nextx,nexty);
/* Save the first fifty points.*/
        count++;
        if(count < 50)
          fprintf(tp,"  %d    %ld   %ld  \n",i,  nextx,nexty);
}
finish:;
        if(argc < 3);
        {   strcpy(filename,"ram:out");
            strcat(filename,argv[1]);
```

Listing Three

```
        a10 = pt1y[0];
        a11 = pt1y[1];
        a12 = pt1y[2];
        a13 = 1;
        a21 = 1;
        a31 = 1;
        h1 = pt2x[0];
        h2 = pt2x[1];
        h3 = pt2x[2];

   deta=a11*a22*a33-a23*a32-a12*a21*a33+a13*a21*a32+a12*a23*a31-a13*a22*a31;
   if(deta==0)
   { printf("determinant = 0\n");
     goto transformation;  }

   deta11=h1*a22*a33-a23*h3+h2*a23-h2*a22*a33+a23*a32*h3-h1*a32;
   deta12=a11*h2*a33-a13*h3+a13*a32*h1-a11*a32*h3+a13*h2*a31-h1*a33;
   deta13=a11*a22*h3-h2*a12+a12*h1-a11*a22*h1+a12*a31*h3-h3*a21;
   a=deta11/deta;
   b=deta12/deta;
   c=deta13/deta;

        h1 = pt2y[0];
        h2 = pt2y[1];
        h3 = pt2y[2];
   deta21=h1*a22*a33-a23*a32*h1+a23*a12*h3-h2*a13*a33+a13*a32*h2-h3*a12;
   deta22=a11*h2*a33-a13*h3-a13*h1*a31+a13*h3*a31-a11*h2*a33+h1*a33;
   deta23=a11*a22*h3-a11*h2*a32+a11*h1*a32-h1*a12*a33+h2*a12*a31-h3*a11;
   d=deta21/deta;
   e=deta22/deta;
   f=deta23/deta;

   printf("%.5f %.5f %.5f %.5f %.5f %.5f
%.0f\n",a,b,c,d,e,f);
   goto transformation;

linear:
      if(cp!= fclose(cp))
      if(cp)
      {
  fprintf(cp,"%5d %4d %5d %4d %5d
%.0f\n",dn,dm,dn,dm,dn,dm);
         fprintf(cp," %5d %5d %5d
\n",xval,xoff,yval,yoff);
         fclose(cp);
      }
      if (!access(mbase,    F_OK|R_OK|W_OK))

}  /* end of main */
```

Listing Four

```
Mission.164
    Listing 4:
                    Tutorial Object File
            IFS_model Weather.dat  Wtunnel
            IFS_decoder Wtunnel mesa
```

Listing Five

```
   Listing 5:
   Points selected during walk-through of the mountain,
   (with labels and comments added).
   These are the points used by IFS_model program to solve
equations
```

```
Use the coefficients a, b, c, d, e, and f of the Affine
transformations that make up an Iterated Function System.

        IN(x,y)                 OUT(x,y)
-------------------------------------------------
                 Curl 1
     0.4    0.0                  0.0    0.0
     0.2    0.0                  0.0    0.2
     0.2    0.2                  0.2    0.2
                 Curl 2
     0.4   -0.4                  0.4    0.6
     0.2    0.0                  0.4    0.8
     0.2    0.2                  0.6    0.6
                 Curl 3
     0.4    0.4                  0.6    0.4
     0.0    0.0                  0.8    0.8
     0.2    0.2                  0.8    0.6
                 Curl 4
     0.4    0.0                  0.8    0.2
     0.0    0.0                  1.0    0.0
     0.2    0.2                  1.0    0.2
                 Curl 5
     0.4    0.0                  0.8   -0.2
     0.0    0.0                  1.0    0.0
     0.2   -0.2                  1.0   -0.2
                 Curl 6
     0.4    0.0                  0.6   -0.4
     0.0    0.0                  0.8   -0.2
     0.2   -0.2                  0.8   -0.4
                 Curl 7
     0.4    0.0                  0.4   -0.6
     0.0    0.0                  0.6   -0.4
     0.2   -0.2                  0.6   -0.6
   xmax=1.   xoffset    ymax=1.    yoffset
    1.000000    0      1.000000     0.0
```

Listing Six

```
Listing 6-1  The first fifty points generated by the
IFS_decoder program during a walk-through of mountain.
Comments and labels added. Points delayed to save space.
In case the calculated points do not give on the screen
that just shows where the points are going so they can
be scaled differentially.

  Transformation      X              Y
-------------------------------------------
     1                 0.0            0.0

     4                 0.2            0.2
     3                 0.4            0.4
     4                 0.6            0.6
     1                 0.4            0.8
     2                 0.6            0.6

     3                 0.8            0.8

     1                 0.6            0.4
     2                 0.8            0.6

     4                 1.0            0.8
     1                 0.8            1.0

     4                 1.0            0.8
     2                 1.2            1.0
```

Please write to:
Laura Morrisson
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Keyboard I/O from Amiga Windows

by John Baez

The Amiga has one of the most powerful and easy to use window interfaces on the market today. It provides different types of objects which fit almost any situation. However, there is one area in which the Amiga gadgets are far from sufficient. This is the area of keyboard I/O.

Keyboard I/O is used to a certain degree in almost all applications. In some applications the Intuition string gadget suffices whereas in others it is clearly not enough. Anyone who has ever intended to write an application which requires substantial keyboard I/O understands what I'm saying.

String gadgets do not permit the use of the up and down cursor keys to move from one gadget to the next. They don't go from gadget to gadget as data is entered in each one. They don't allow things such as right justification or automatic edit checks when data is entered. Most of all they are very limited as far as the different types of data which can be entered. They offer no options at all as to where in memory to store the data once it has been entered.

These limitations make the use of string gadgets for keyboard I/O very awkward and cumbersome for both the user and the application programmer. In this article we'll create a new type of window object called an IOField which provides a viable alternative for keyboard I/O.

The features of IOField implemented in the code that accompanies this article include the capability to enter data that accommodates short and long integers, floats and doubles, character strings composed of numeric digits and character strings made of any characters. You can also define your fields as display only(OUTONLY), non-display data entry(INONLY), display and entry (OUTIN) or display after entry (INOUT).

Fields can be displayed with a rectangular border (BORDERED), a double border (HILITE) or underlined (UNDERLINE). You can use the text and background colors of your choice.

Positioning into a field can be controlled either by keyboard or mouse. The data resulting from keyboard I/O is stored at a predesignated location of your choice.

THE DESIGN OF IOFIELD

In planning how we want an IOField to look and behave there are many possibilities. The code presented here is but a foundation on which a very sophisticated, yet easy to use, keyboard I/O facility can be implemented.

Just like all other Intuition window objects, the IOField has a structure which is used to create a linked list of IOFields with the desired attributes for the application at hand. A function is then provided which takes care of manipulating the linked list structure and dealing with the I/O from the designated window. When keybaord I/O is ended or when an event not related to IOField is detected, the function returns the event to the caller.

Listing 1 illustrates the IOField structure. To give you an idea of how IOField works, here's an explanation of each of the fields in the structure.

The NextData field is just a pointer to the next IOField structure in the list or NULL if there are none. LeftEdge and TopEdge identify the position of the top, left hand corner in the window of the rectangle in which a field is displayed.

The Flags field is used to specify the attributes of an IOField. In this implementation those values are outlined in Table I. Later we'll see the effects of these attributes and how you can add additional ones.

The FrontPen, BackPen and DrawMode fields indicate the color of the text, the background of the text and the drawmode for using these fields. These drawmodes are the same as those used by other Intuition objects such as IntuiText.

The Length field indicates the length of the field (in characters) as it will appear in the window. The Data field is a pointer which points to the location in memory from which the field's value is read and to which it is written. This is usually a pointer to a field in a data structure. We'll see how those work later.

The EdFunc field points to a programmer-supplied edit routine which will be called whenever a value is entered or modified in the field. The last field is for programmer use and can point to anything the programmer may wish to associate with the field. The keyboard I/O routine ignores this field.

The internal functionality required to support the IOField structure has 6 distinct functions. It displays the IOFields with their initial data values, reads window events, converts rawkeys, performs a specific action based on the key pressed, navigates through fields either by mouse or keyboard and optionally calls an edit routine at the completion of each field's data entry. Let's take a closer look at these functions and their implementation.

The first function is to display the fields with their current values in the designated window. This is done by the displayIOData() function (Listing 3). First the field values are converted to character strings, by calling convfield(), then they are displayed as specified by the Flags field of the IOField structure. See Table I for a list of the flags and their meaning.

Once the fields are displayed we wait for an IntuiMessage to come in for the designated window. When we get one we first determine if it is of interest to the IOField functionality. If it is, we proceed to process it. Otherwise we pass the IntuiMessage back to the calling application. These activities are performed by the getkey() function (Listing 3).

The messages intercepted are RAWKEY and MOUSEBUTTONS. Raw keys are converted to an unsigned longword representing the key hit. The "normal" keys are assigned their ASCII code whereas function keys are returned as a predetermined value. The value for some common keys are given in the header file "keys.h" shown in Listing 2. The conversion is performed by the function whichkey().

MOUSEBUTTONS are intercepted only if they are caught at a location within the rectangular display area of any of the IOFields. Otherwise they are returned to the calling application. The find_field() function (Listing 3) searches the IOField linked list with the mouse coordinates to see if they are within an IOField.

The function performed on RAWKEYs depends on the key pressed. If the key has no meaning for IOField processing, it is passed back to the calling application. The getkey() and getData() functions in Listing 3 show the keys accepted in a case structure. The getkey() function manages keys which work within the current field whereas getData() handles keys which have meaning in the context of the entire window.

When each field is completed, an optional, user supplied, edit function is called if present. This function can perform specialized edits on the field. An example, which can be used as a template, is presented in Listing 4. If this function returns a non zero value the screen "beeps" and the cursor doen't advance to the next field. The application will not let the user past this point until the edit function returns a zero (meaning the edit passed).

The arguments passed to the edit function are a pointer to the current Window, pointer to the current IOField and a pointer to the top of the IOField linked list. You can do just about anything within this function, even make a call to getData().

Table One

IOField attributes

Flag name	*function*
OUTIN	The field will display it's current value and then permit keyboard I/O to modify it.
INOUT	The field's initial value will not be dis-played. Keyboard I/O is permitted to place a value into the field.
OUTONLY	The field's current value will be displayed but cannot be modified.
INONLY	The field's value is not displayed at all. Keyboard I/O is permitted to modify it.
INTEGER	The field value is stored in a 2 byte (short) integer.
LONG_INT	The field value is stored in a 4 byte (long) integer.
FLOAT	The field value is stored in a 4 byte floating point field.
DOUBLE	The field value is stored in an 8 byte floating point field.
DIGITS	The field value is stored as a nullterminated string of numeric digits.
ANYCHAR	The field value is stored as a null terminated string.
BORDERED	The field is displayed with a border forming a rectangle around the I/O area using the FrontPen color.
HILITE	The field is displayed with a double border forming a rectangle around the I/O area using the FrontPen color.
UNDERLINE	The field is displayed with an underline using the FrontPen color.

Listing 5 is a sample program. The application is the outline of a simple mailing list which shows the use of different IOField attributes. It illustrates how the IOField structure is initialized and how the getData() function is used to process events in place of the more typical Wait()/GetMsg() combination. It also includes a handy window() function which permits you to open a window without having to supply a NewWindow structure.

Lines 81 through 103 define the IOFields and their support structure. Notice how the mailing list data is kept in its own, user defined, structure (called mailData) separate from the IOField structures. In this fashion the application doesn't have to deal with the IOFields for anything. This permits the programmer to concentrate more on the job at hand.

Lines 143 through 172 illustrate the main loop of the function. Here you can see how the getData() function is used to get events from the window. The events are processed just as if they were received from a GetMsg() function. Please note that your application may receive MOUSEBUTTONS, RAWKEY and INACTIVEWINDOW events from the getData() function even if they haven't been specified in your IDCMPFlags. Therefore you should always supply at least a stub for these IntuiMessage types.

If you compile and link the code using the commands in Listing 6, you'll see how the fields work from the application point of view. You can play with the attributes to get a feel for how they work in different combinations.

ENHANCING THE IOFIELD OBJECT

As I said before, the code presented for the getData() function is a foundation on which a very powerfull keyboard I/O facility can be built. For example, you can add a context-sensitive help facility which would display help text associated to each individual field whenever the HELP key is pressed. You can add field types such as date, time, currency, and others. You can add range checks and data selection from a preset list of values. Another nice feature to add would be edit masks which permit the field value to be displayed in a more legible format. I'm sure you could think up some more.

Facilities such as this coupled with other existing facilities make for easy to write, professional-looking applications.

Usually adding a feature will mean either adding another flag or adding fields to the IOField structure. When implementing features, always try to keep in mind ease of set up and use. Adding features attached to keys is usually just a matter of adding the key to the RAWKEY case statement in either the getkey() or the getData() function.

Well, I hope you find the code useful. If I get enough requests, I'm willing to do a follow-up article with additional features. In an upcoming article, I'll be providing you with a handful of neat little functions which can be combined to produce impressive windows with very little programming effort. I'll show you how to create sophisticated edit functions. I'll be bundling them up and placing them into a shared library which you'll then be able to access from almost any Amiga supported language.

Table Two
Keyboard functions

Key name	*function*
Left	moves cursor left one character unless it is at the start of the field.
Right	Moves cursor right one character. At end of field advances to the start of the next field.
Up	Moves cursor to the start of the preceding field.
Down	Moves cursor to the start of the next field.
HOME &Pageup	Moves cursor to the start of the first field.
Forward tab	Moves cursor to the start of the next field.
Pagedown	Moves cursor to the start of the last field.
End	Completes data entry and returns to the application.
Back space	Deletes the character to the left of the cursor unless the cursor is at the start of the field.
Delete	Deletes the character at the cursor position.
Insert	Inserts a space for a character to be typed into.
Return	Advances to the next field.
Escape	Clears the field of its current value and places the cursor at the start of the field.

Listing 1: getdata.c

```
```

Listing 2: mailer.c

```
```

Listing 3: edit functions.c

Listing 4: keys.h

Listing 5: getdata.h

Please write to:
John Baez
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

GOTTA GETTA GUIDE!

Looking for a specific product for your Amiga but you don't know who makes it? Want a complete listing of all the Fred Fish software available? Just looking for a handy reference guide that's packed with all the best Amiga software and hardware on the market today?

If so, you need *AC's GUIDE for the Commodore Amiga*. Each *GUIDE* is filled with the latest up-to-date-information on products and services available for the Amiga. It lists public domain software, user's groups, vendors, and dealers. You won't find anything like it on the planet. And you can get it only from the people who bring you the best monthly resource for the Amiga, *Amazing Computing*.

So to get all this wonderful information, call **1-800-345-3360** today and talk to a friendly Customer Service Representitive about getting your *GUIDE*. Or, stop by your local dealer and demand your copy of the latest *AC's GUIDE for the Commodore Amiga.*

List of Advertisers

MOVING?

SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

Amazing Computing Subscription Questions
PiM Publications, Inc.
P.O. Box 869
FallRiver, MA 02722

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing

AC's TECH Disk
Volume 2, Number 2

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with 'look' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PIM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 2140
Fall River, MA 02720-2140

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that activate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PIM Publications, Inc. their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This statement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright © 1990, 1991 by PIM Publications, Inc. and may not be duplicated in any way. The purchaser however is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties on your computer equipment. PIM Publications, or any of it's agents, is not responsible for any damages incurred while attempting the projects.

COPPER PROGRAMMING WITH HISOFT BASIC

WOULD YOU BELIEVE AN 800-COLOR WORKBENCH?

by Robert D'Asto

What Is the Copper?

One of the hardware items which makes the Amiga such a graphics powerhouse is known as the graphics coprocessor or Copper. The Copper is a secondary processing unit residing in one of the Amiga's custom chips which relieves the main processor (the 680x0) from certain tasks related to Amiga graphics rendering, thus providing a substantial boost in overall graphics display speed. It can control or modify the screen's display on a very low level, capable of bringing about changes on the screen at the speed of the monitor's scanning beam. Among other things, the Copper can change the contents of the color registers (as the PALETTE statement does) on each horizontal line of the display as it is scanned or "drawn" by the monitor. This is how NewTek's Dynamic HiRes Mode and other similar display modes can take a hi-res screen with a depth of four which is normally capable of only 16 colors, and turn it into a 4096-color display. Each horizontal line of the screen can display only 16 colors, but those 16 colors are being redefined on every scan line, thus making thousands of different colors possible on the entire screen.

The Copper is usually controlled entirely by the operating system but the Amiga designers also provided means for placing the Copper under application control, allowing programmers the opportunity to directly take advantage of its power over the display. This can be done with BASIC but it requires an understanding of certain elements of the Amiga graphics display system.

A Quick Overview

Like most processors, the Copper is a device which can perform certain data operations when it is provided with coded instructions and the data itself to manipulate. The instructions are referred to as the processor's instruction set and usually consist of "numbers" which the unit has been designed to recognize as commands to perform certain operations. As an example, the Motorola 68000 chip has an instruction set of around 300 commands which it can recognize and perform. Fortunately, the Copper's instruction set is much smaller, comprising only three instructions. This may not seem like much at first, but some startling effects can be produced by programming the Copper, as can be seen from the listing at the end of this article: a small program which can turn on and off an 800-color Workbench screen without adding a single bitplane to the display. Of course, many other effects are possible with Copper programming but I'll leave that matter up to you.

A set of instructions for the Copper is referred to as a Copper List. This list has an exact format which, if you are an Assembly Language programmer, can be "made from scratch" and given to the Copper for processing. The details for this method of building a Copper List can be found in the *Amiga Hardware Reference Manual*, published by Addison-Wesley, but there is a much easier way to create Copper Lists through the use of system library routines and is described below.

Once created, the memory address of the Copper List is "plugged into" a certain block of data maintained by the system known as a ViewPort Structure and the system is asked to reconfigure itself according to this new Copper List. From then on the display is under control of the new Copper instructions provided by the programmer.

The ViewPort

A ViewPort, in Amiga system parlance, is a rectangular section of the display with certain, definite restrictions. The entire display can be divided into two or more ViewPorts but they must be above or below one another, never side by side. ViewPorts must also never overlap, being separated by at least one horizontal line of pixels. Usually, when programming with BASIC, the displays created have only a single ViewPort which occupies the entire monitor screen. It is possible, though, to create a display which is divided into two or more ViewPorts, each having different depths, resolutions, and color definitions. This isn't done very often, but it is possible on the Amiga.

The system keeps track of each ViewPort (usually only one) with a block of data known as a ViewPort Structure, which has an exact form. Since the term structure may be unfamiliar to you, I'll cover that ground first. A structure is simply a block of data residing in RAM and consisting of individual values which are usually one, two, or four bytes in length. Each of these values contained in the structure is called a field. The values contained in each field can change, but not their sequence within the structure. Structures are very handy for storing a bunch of data of different types and, since the sequence of the data remains the same, the values contained in individual fields can be retrieved or changed by counting forward the necessary number of bytes from the beginning of the structure to the desired field. The position of a field in relation to the start of the structure is referred to as the field's offset.

Each line in the definition below represents a single field of the ViewPort Structure and is identified with a name for reference, its size in bytes, and a brief description of the value contained in that field. This structure is 40 bytes long.

```
ViewPort Structure:
Next    (4) - address of next ViewPort, if any
ColorMap (4) - address of data describing colors used
DspIns  (4) - address of Copper List, display info
SprIns  (4) - address of Copper List, Sprite info
ClrIns  (4) - address of Copper List, Sprite info
UCopList (4) - address of User Copper List
DWidth  (2) - width of the ViewPort
DHeight  (2) - its height
DxOffset (2) - x position of upper left corner
DyOffset (2) - y position of upper left corner
Modes   (2) - display mode used
reserved (2) - nothing yet
RasInfo (4) - address of another structure called RasInfo
```

Only four of the above fields are relevant to the matter at hand, programming the Copper. These are the fields DspIns (Display Instructions), SprIns (Sprite Instructions), ClrIns (Color Instructions) and, most importantly, the UCopList field which is the address of the User Copper List. The user, in this case, is the programmer. All four of these fields contain pointers to (addresses of) Copper Lists, but it is the User Copper List which comprises the customized Copper instructions needed to put the Copper under program control. The three other Copper Lists are maintained by the system.

Creating the User Copper List

Making our own Copper List is done with the help of a few system library routines from the exec and graphics libraries. This requires a familiarity with the use of the BASIC LIBRARY and DECLARE FUNCTION statements as well as bmap files. If you are unfamiliar with these subjects more information can be obtained from the AmigaBASIC manual, data on the Extras Disk, and numerous *Amazing Computing* articles. To compile the listing provided, you must have copies of the files "intuition.bmap," "graphics.bmap," and "exec.bmap" in the LIBS directory of your system disk at compile time.

First, a special structure must be created called a UCopList Structure. This is a very simple structure with a length of 12 bytes, containing three 4-byte values which are each addresses of other, related structures. The details of these other structures don't concern us here, since we don't even need to provide the addresses. All we really need to do is set aside an area of 12 bytes in RAM for the UCopList Structure and let the library routines do the rest of the work.

Allocating memory for use as structures can be done with the exec library routine AllocMem. This routine returns a long integer value (the address of the allocated memory) so requires a DECLARE FUNCTION LIBRARY statement as shown near the top of the listing. The calling syntax of the function is:

```
Address&=AllocMem& size&,opt&
```

The size& parameter is the amount of memory you want in bytes and opt& is a value which represents one or more options available in the allocation of the memory. The possible values are:

```
0 -> no options, any memory
1 -> nonrelocatable memory
2 -> CHIP RAM, memory accessible by custom chips
4 -> FAST RAM, memory inaccessible by custom chips
65536 -> Clear the memory area upon allocation
```

These options can be used singly or in combination by adding the appropriate values and using the sum as the opt& parameter when calling AllocMem. An example of the use of this function can be found within the AllocCopper sub in the listing, which allocates the 12 bytes needed for the UCopList Structure. When AllocMem executes, it returns the address of the allocated memory in the variable provided, Address& in this case. If AllocMem fails in its allocation of the requested memory (usually because there is insufficient free memory available) it returns a value of zero. The program should always check to see that the returned value is not zero before proceeding, otherwise a system crash could result if the program tried to do something with a block of allocated memory that doesn't exist.

Creating the User Copper List itself is done with the use of three graphics library routines. Calling these routines causes certain Copper instructions to be added to the list.

The first of these routines is called CWait, meaning "Copper Wait." It is a command for the Copper do nothing until the monitor's video beam reaches a certain x/y coordinate. The routine is called like this:

```
CWait& UCopList&,y%,x%
```

The first parameter is the address of the User Copper List Structure which was allocated earlier with AllocMem. We have this address because AllocMem returned it after its execution. The next two parameters represent the screen coordinates for which the Copper is to wait, that is, the screen location which the video beam must reach before the Copper can go ahead with its next instruction. Notice that the y coordinate comes first, the reverse of the usual. Calling this routine will add an instruction to the User Copper List for the Copper to wait for the specified screen location to be reached by the video beam.

The next Copper routine is called CMove. It places a command in the Copper List telling the Copper to write a specified 16-bit integer value to a particular location in memory. It is similar to the POKEW command in BASIC. The memory location written to can be anywhere except the lowest 16 or 32 bytes (depending on the use or non-use of an option, which is beyond the scope of this article). The CMove routine is called like this:

```
CMove& UCopList&,register&,value%
```

The first parameter is the same as for CWait, the address of the Copper List Structure. The register& variable is the memory address where the value is to be written, and value% is the 16-bit value to be written to that location.

Why use the Copper just to POKEW values into memory? The intended destination for these values is not just any memory

location; it is a special area of RAM known as the hardware registers. This is a zone of memory used by the custom chips and parts of the operating system to store special values related to the current operating state of the machine. Of particular interest are the color registers, a 64-byte strip of memory which holds the current color data for the Amiga's 32 basic colors. Each color is represented by a 2-byte value which gives the red, green, and blue content of the displayed hue. These values range from zero to 4095 or, in hex, &H000 to &HFFF. It's easier to use a three-digit hexadecimal notation for these colors because this illustrates the actual color being represented. Each of the three hex digits represents the red, green and blue content in that order. For example, the value &H408 would mean 4/15 red, 0/15 green, and 8/15 blue.

When calling the CMove routine to change color registers the values to use for the register& parameter are the even values from 384 to 446. These are the memory locations for the color registers. Color register zero (background) would be 384, register one would be 386, register two is 388, and so on to color register 31 which is 446. By the way, these are not absolute addresses; they represent offsets from the absolute address &HDFF000.

There is a third Copper List routine called CBump. This is used after each call to CWait or CMove to indicate that the next Copper instruction is to be added to the list after the previous one. It's like pressing the return key after entering a line of BASIC code. If not used, then each instruction would be written on top of the last. To make things simpler in the listing, one places the CWait and CMove calls in subprograms called CopperWait and CopperMove, each of which also contains a call to CBump. This saves a lot of typing. The "waits" and "moves" can then be implemented by calling the subs like this:

```
CopperWait y%,x%
CopperMove register&,value%
```

The address of the User Copper List is declared as a SHARED variable in both subs so it does not have to be included as a parameter each time the subs are called.

By now you should be getting the idea of how Copper programming is done. Using the CWait and CMove routines within the CopperWait and CopperMove subprograms, we can provide instructions for the Copper to change color registers at any desired screen location. If, for example, we wanted the background color of the top horizontal scan line to be black and the background color of the next line down to be white we would code:

```
'wait for upper left corner of screen
CopperWait 0,0
'change COLOR 0 to black
CopperMove 384,&H000
'wait for next line down
CopperWait 1,0
'now change COLOR 0 to white
CopperMove 384,&HFFF
```

Actually, the above code would make the background color of the top line black and it would be white on all the remaining lines unless we added further instructions to change it to something else after the second scan line. Notice that, in the second call to CopperWait, the y coordinate comes before the x coordinate.

It's important to note that in the above example we are not simply drawing lines of different colors onto the background of the screen. We are *redefining* the background color register for that line. If you can imagine being able to specify PALETTE statements for every horizontal line on the screen's display, you'd have the idea.

Since we can change the color registers on every horizontal scan line, a Copper List could require a lot of instructions. Much typing can be saved by using FOR/NEXT loops for this purpose as is done in the "Main_2:" section of the listing.

Like most processors, the Copper is a device which can perform certain data operations when it is provided with coded instructions and the data itself to manipulate.

There Is CMore

There are two other subs in the listing: AllocCopper and InitCopper. The AllocCopper subprogram allocates the 12 bytes needed for the UCopList Structure using the exec AllocMem function as described above. The address of this allocated memory is returned in the variable UCopList& which is SHARED by all the other subs. It also contains a mysterious POKEL statement which is explained a little later.

The InitCopper subprogram does three things. First it calls the CopperWait sub like this:

```
CopperWait 10000,256
```

What is this? How can the Copper wait for a screen position with these coordinates? It can't—well, it can wait but it better not hold its breath because the video beam will never reach the 10,000th horizontal line as it doesn't exist on the Amiga, not yet anyway. Using this wait instruction is simply the normal programming method for ending the Copper List instructions. It closes the instruction list.

The next line in the InitCopper sub places the address of the User Copper List in the correct field of the ViewPort Structure.

This is the UCopList field which is 20 bytes from the top of the structure. The address of the ViewPort Structure was obtained earlier and placed in the variable VP& in the "GetPointers:" portion of the listing. This is accomplished with a routine from the intuition library called ViewPortAddress. ViewPortAddress is a function which returns a long integer value so it also requires a DECLARE FUNCTION LIBRARY statement. This routine needs only one parameter: the address of an existing Window Structure. As you may have guessed or already know, a Window Structure is a structure for defining an Intuition window, but more on this later, too.

The last line of the InitCopper subprogram calls an intuition library routine called RethinkDisplay. This routine requires no parameters; just call it by name. The address of a User Copper List has been provided in the ViewPort Structure and RethinkDisplay will see it and incorporate its instructions into the screen's display.

The Tricky Stuff

What we have here is a noninteractive, toggling program. Run it once and it does something to the Workbench display. Run it again and it undoes what it did the last time it ran, returning the display to normal. The problem is: How does it know when to "do" or "undo"? A further problem is: When it "does," it causes memory to be allocated which must be freed up when it later "undoes"; otherwise, it will eat up memory every time it runs, a rather annoying attribute for a program. So, when it "undoes" how will it find the memory that was allocated the last time it ran so it can then be freed up?

A ViewPort, in Amiga system parlance, is a rectangular section of the display with certain, definite restrictions. Usually when programming in BASIC, the displays created have only a single ViewPort.

First let's look at how much allocated memory is involved. We used the AllocMem routine to allocate 12 bytes for the UCopList structure. That's not so bad. I suppose we could just call it negligible and forget about it. Even if the program were run 100 times, only 1.2 K of memory would be lost. Unfortunately there is more to it than this. The User Copper List itself takes up memory. Each instruction in the list uses four bytes. The FOR/NEXT loop in the "Main_2:" section of the listing goes around 200 times, adding five instructions to the User Copper List each time. That's 1,000 instructions times four bytes or 4K of memory just for the Copper List. There are also structures initialized by the system that come into play. The bottom line is, it just wouldn't do to forget about freeing up that much memory.

Let's attack the first problem: How does the program know to "do" or "undo" the display? This one is pretty simple, really. The program leaves a "note" tacked to the Workbench screen whenever it "does" the display and removes it whenever it "undoes" the display. Each time the program starts, it first looks for the note. If the note is there, it "undoes" the display; otherwise, it "does" it.

I'll have to back up a bit to explain what I mean. All the graphic objects which are part of the Amiga operating system have structures which define them. Windows, menus, gadgets, screens and so on all have special structures in RAM associated with them. The Screen Structure is rather large, over 300 bytes in length, and contains a great deal of information, including entire substructures. There is one particular field in the Screen Structure called the UserData field which can help us with this problem. This field is four bytes long and is located 338 bytes from the top of the Screen Structure. It's there for the programmer to do with as he chooses. Any value can be placed there and the system will maintain it. This is where the program leaves its "note." This is in the form of an arbitrarily selected value, &HCCCC, which is placed in the UserData field of the Workbench Screen Structure by the AllocCopper subprogram in the listing whenever the program creates the User Copper List. The UserData field is cleared to zero whenever the program "undoes" the display. When the program starts up, it looks in this field to see if the code value is there. From that point it branches to the appropriate operation.

There is one little wrinkle to this. How do we find the Workbench Screen Structure? Intuition comes to the rescue here in the form of an intuition library function aptly called GetScreenData. This is how the function is used:

```
success%=GetScreenData&
buf&,size%,type%,screen&
```

The success% variable will be assigned one of two possible values when the function executes. It will be false (zero) if the function failed or true (-1) if it succeeded. The success variable can be tested after calling the function to see what happened and the program can branch as appropriate. In the listing provided the program aborts if the function fails. The buf& parameter is the address of a buffer to hold the data that GetScreenData provides. You can use an array for this purpose and pass the value VARPTR(array(0)) for the buf& parameter. Size% is the size of the buffer in bytes. The type% parameter is one of two values. If you're after the Workbench screen use a value of one (&H1), otherwise use a value of fifteen (&HF). The last parameter is ignored by the GetScreenData if you have requested data on the Workbench screen so it can be zero in this case (as it is in the listing); otherwise, it is the address of the Screen Structure being sought. The function will read all or part of the Screen Structure into the buffer you have provided, the amount read depending on the size% parameter. GetScreenData also requires a DECLARE FUNCTION statement.

Before calling GetScreenData, a buffer is created by DIMing long integer array of sufficient length and its address is passed as given above. For our purpose we need only the first eight bytes of the Screen Structure, as bytes 5 through 8 contain the data we're after: the address of the Workbench screen's window. This is, of course, the address of another structure, a Window Structure. Once we have this we can find the address of the ViewPort Structure (to which we will later attach the User Copper List) by calling the ViewPortAddress function described above. (The ViewPortAddress function requires the address of a Window Structure to do its job.) While we're at it, we can also find the address of the Workbench Screen Structure itself by PEEKLing at an offset of 46 bytes from the top of the Window Structure, as this is where that address is located. This allows us to locate the UserData field of the Screen Structure to see if it contains the "note" or not, so the program will know which way to go. All of this is done in the first few lines of the "GetPointers:" portion of the listing.

Dismantling the Display

All of the freeing up of previously allocated memory is done with a graphics library routine called FreeVPortCopLists. There are two other, similar routines in the graphics library called FreeCopList and FreeCprList. The difference has to do with the fact that there are different types of Copper Lists. According to all the system documentation I have seen, none of these routines is designed to free up the memory specifically and only associated with a User Copper List. The safest bet seems to be the FreeVPortCopLists routine. Notice the "s" at the end of its name. This routine deallocates the memory for all the Copper Lists contained in the ViewPort Structure at once. It first looks at the four fields in the ViewPort Structure which contain Copper List Pointers and works from there, freeing up all the allocated memory associated with them. The trouble is we don't want all the Copper Lists dismantled, just the User Copper List. We want to leave the Copper Lists associated with the normal Workbench display alone so that the original display is restored. To solve this problem, I've come up with a little trick to fool FreeVPortCopLists into freeing up only the User Copper List and the trick works. The program first saves the contents of the ViewPort Structure fields which contain the addresses of the three other Copper Lists. (See the DspIns, SprIns and ClrIns fields in the definition of the ViewPort Structure above.) After their addresses are saved, these three fields are then cleared to zero and FreeVPortCopLists is called. Seeing nothing in the first three Copper List fields of the ViewPort Structure, the routine ignores them. Since the UCopList field is the only one containing an address, the User Copper List memory is freed. After the routine finishes, the values previously held in the three other Copper List fields are returned and the original display is restored with a call to Intuition's RethinkDisplay routine.

Compiling and Running

The listing was written for the HiSoft BASIC Professional compiler version 1.05. The compiled program, which is on the companion disk for this issue, was compiled to stand-alone form, so it can be moved around without fear of losing support files. The resulting file size is about 16K, including the linked HiSoft library. If you are typing in the program and compiling it yourself, you may have to change the very first line of code to suit your hardware configuration. This line contains the "F" option instruction "FDF1:WB800" which outputs the executable to a file on DF1:, which won't work if you don't have a DF1: mounted. Running the program is done either by double clicking its icon or by direct call via the CLI/Shell. The resulting executable file, called WB800, is rather startling: an 800-color Workbench screen! The more windows and icons on the screen, the more colors will be visible. Each of the four Workbench colors is redefined on each of the 200 horizontal scan lines, so moving an icon or window vertically causes its colors to change. Running the program again restores the original Workbench display. You can also see by the memory counter at the top of the screen that all of the memory consumed by the customized display is released when the display is restored, so the program can be run as many times as desired.

Copper Applications

Normally you wouldn't have to worry about freeing up the Copper List as is done in the Main_1: part of the code because the system will do this when the screen is closed. This listing is a special application for an unusual situation: providing a User Copper List for the screen of a currently-running program, Workbench. The usual application would be to create a custom SCREEN and WINDOW, provide a User Copper List with some variation of the Main_2: section of the code and then close the screen upon exit from the program. Closing the screen will free up the User Copper List.

Copper programming is a great way of providing additional color to a display without gobbling up chip RAM by adding bitplanes (depth) to a screen. Multi-colored backgrounds, text, or graphic objects become easier to provide, especially with larger programs which already consume quite a bit of RAM due to the hefty files output by the compiler. Theoretically, a screen with only a single bitplane (depth of one) could display 400 different colors at once using this method!

Feel free to experiment with the code. It's interesting to see the different effects possible.

☑

All listings for Copper Programming can be found on the AC's TECH Disk

Please write to
Bob D'Asto
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

MenuScript

Creating commercial-quality menus with this easy-to-use language

by David Ossorio

Introduction

This is MenuScript, a small high-level language for designing text menus on the Amiga. MenuScript is both a language and an interpreter, easy to learn and easy to use, which lets you describe the appearance of your menu with simple ASCII scripts, using a minimum of effort. The interpreter displays your completed menu and writes its source code to the file of your choice in any of C, assembly language, or AmigaBASIC. Just compile and link. What used to take days can now be done in minutes.

Motivation

MenuScript is unique among Intuition utilities. Its design was guided by two philosophies: a language for describing text menus should be simple and easy to use, but control over fine detail should not be sacrificed for such ease-of-use. More about the second point later, but first, what is meant exactly by ease-of-use?

Suppose we want plain vanilla menus with no fuss. Then MenuScript should be smart enough to fill in the details in a way that makes the menus look good, even though we haven't specified anything but the actual text for the menus. In other words, MenuScript should have reasonable defaults which do not have to be continually reset after each use.

This may seem obvious, but consider a commercial program like *PowerWindows*, which requires you to perform a long sequence of clicks and typing to construct plain text menus, and even then the results are often not what you had in mind. The GUI technique that works so well for designing gadgets and images is not necessarily the most efficient way to design a text menu. It would seem more natural to describe a text menu with actual text, using a language which hides the most tedious details of constructing the source code for that menu.

As an example, here's how you would construct a simple menu in AmigaBASIC:

```
MENU 1,0,"Project"
MENU 1,1,"New"
MENU 1,2,"Open"
.
.
.
```

This is certainly much more intuitive and simpler than the corresponding C code would be for the same menu. We have specified only minimal information, and AmigaBASIC has done the rest.

But there is a steep price we pay for this ease-of-use in AmigaBASIC, namely that we are able only to display the most general type of text menus. There are no provisions for specifying command key equivalents, which are present in virtually all Amiga software. We can't even construct submenus in AmigaBASIC without resorting to calling the operating system routines, which defeats the whole purpose of using AmigaBASIC in the first place. Clearly this is an unacceptable compromise, and it brings us to the second design philosophy behind MenuScript. A language for designing text menus should allow you to create complex non-standard text menus. It should allow you to override any default values it uses and specify the exact appearance of the menu. In short, anything you could do if you wrote the source code yourself you should be able to do with that language. Obviously this is not a small requirement, but you will see that MenuScript comes pleasantly close to this ideal.

How to Start It

MenuScript is called from the cli by typing the following:

ms [>redirection file] [-dplabcz] inputfile.menu

MenuScript sends all of its output to stdio. If you don't want the output printed on the screen (which is usually the case), you must redirect it by typing ">redirection file". For example, "ms >df1:MyMenu.h MyMenu.menu" reads the file "MyMenu.menu" and sends the output to "MyMenu.h"

Additionally, MenuScript recognizes the following optional switches:

-d This tells MenuScript not to display your menu. By default MenuScript lets you see the completed menu before it writes out the source code. Using the '-d' switch overrides this.

-p This tells MenuScript not to write the source code for the completed menu. Useful if you're experimenting and only want to see the menu.

-l [letter el] This tells MenuScript not to display the line numbers when reading the input file. Normally when you run MenuScript you get a message like this:

```
MenuScript Version x.x
Reading... [line number]
```

where [line number] is the line of the input file that is currently being read. If you're really in a hurry, you can speed things up slightly by turning off the [line number] display in this manner.

-z If you use this switch, MenuScript prints a list of its default settings and terminates.

-a This tells MenuScript to create assembler source code.

-b This tells MenuScript to create AmigaBASIC source code.

If you don't specify otherwise, you get C source code; the -c switch is redundant. MenuScript can only output one type of source code at a time, so only one of C, Assembler, or AmigaBASIC may be specified.

How To Use MenuScript

Before we dive into a precise definition of MenuScript syntax, look at the following file written in the MenuScript language:

```
;=============================================
;Here is a short file that describes a relatively
simple menu.
;=============================================
    .menu "Project"
    .item "New" key N
    .item "Open" key O
    .item "=> Print"

    .subi "Draft" checkable check
    .subi "NLQ" checkable
    .item "Quit" key Q
;=============================================
```

Notice that MenuScript is compact without being cryptic, so you won't have to spend a lot of time learning its syntax. Also, each line of a MenuScript input file can be divided into the following simple elements:

1. Comments

Any text following a ';' or '*' is treated as a comment. Comments end at the end of a line, and a completely blank line is treated as a comment.

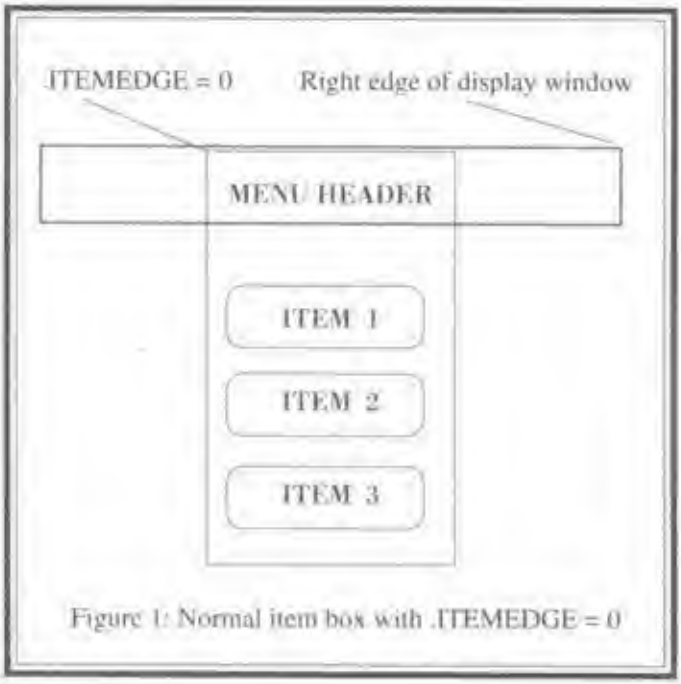


Figure 1: Normal item box with .ITEMEDGE = 0

2. Commands

In MenuScript, commands always start with a period (.) and are used to declare menus, items, and sub-items. There are also commands to change default values used in determining how the menu will look.

3. Parameters

These are sometimes optional arguments which always follow commands. They allow you to give the MenuScript interpreter more specific information about the appearance of your menu.

There are only a few things to keep in mind when writing a file in the MenuScript language. First, remember that MenuScript is a scripting language. Only one command is allowed per line, but that command can be placed anywhere on the line. Parameters which follow a command must be on the same line as the command, and they can be separated by commas or spaces or both. Also, commands and parameters may be written in upper or lower case, or both.

That's it. Now we take a detailed look at all of the commands and parameters which make up the MenuScript language. You'll find that MenuScript is rich enough to express just about any type of text menu you might want to use in your software application, and that even very complex menus can be constructed in minutes.

Commands

Command .MENU
Format .MENU <name> [menu option]*
Template .MENU "MenuName" OFF
Purpose To begin a new menu.

([option]* means that zero or more options may follow this command.)

The .MENU command sets the title bar of a menu. For example, for a "Project" menu, you would write:

```
.menu " Project "
```

Any .ITEM commands encountered after the .MENU command would naturally belong to this menu. To start a new menu to the immediate right, simply repeat the .MENU command.

Currently the only parameter which may be specified after the .MENU command is OFF, which will initially ghost all items and sub-items which belong to this menu. Using the OFF parameter is equivalent to clearing the MENUENABLED flag of the Menu structure for this menu.

Command .ITEM
Format .ITEM <itemname> [item option]*
Template .ITEM "ItemName" [CHECKABLE,CHECK,
TOGGLE, OFF,BOX,COMP,NONE,
SELECT <name>,EXCLUDE <item number>*
END,
KEY <command key>]
Purpose To define a menu item.

.ITEM sets the text for an item belonging to the latest .MENU. Any sub-items (.SUBI) encountered between here and the next instance of .ITEM will belong to this item. So continuing our example from above, suppose that we wanted our "Project" menu to contain four items. We would then write:

```
.menu "Project"    ;this is our project menu...
.item "Open"
.item "Save"
.item "Print"
.item "Close"
```

[item options] are explained when we look at parameters.

Command .SUBI
Format .SUBI <name> [item option]*
Template .SUBI "Sub-item Name" [CHECKABLE,CHECK, TOGGLE, OFF,BOX,COMP, NONE,SELECT <select name>, KEY <command key>, EXCLUDE <item number>* END]
Purpose To define a sub-item.

.SUBI sets the text for a sub-item belonging to the latest item. In our example, if we wanted the "Print" item to have two sub-items, we would write:

```
.menu "Project"
.item "Open"
.item "Save"
.item "Print"
.subi "NLQ"
.subi "Draft"
.item "Quit"
```

[item options] are identical to those for the .ITEM commands and are explained when we look at parameters.

Command .NAME
Format .NAME <name>
Template .NAME "MyMenu"
Purpose To specify the variable name of the pointer to your menu strip.

In any program that uses menus, you must call the Intuition functions SetMenuStrip( ) and ClearMenuStrip( ) to set and clear your menu strip. These functions require a pointer to a Window structure and a pointer to a menu structure as input. When you use the .NAME command (i.e., .NAME "MyMenu"), MenuScript puts the following line in the menu source code:

```
struct Menu *MyMenu = ..; /* Create a global
variable MyMenu */
```

The variable 'MyMenu' points to your menu strip, so you can easily write:

```
SetMenuStrip(window,MyMenu);
```

in your application program.

If you do not use the .NAME command, MenuScript assigns a default name, 'MText', to your menu, and in that case the menu could be displayed in your application by writing:

```
SetMenuStrip(window,MText);
```

Command .FONT
Format .FONT <fontname> <height>
Template: .FONT "yourfont.font" <height>
Purpose To construct a menu using a custom font.

.FONT allows you to easily construct a menu using a custom font. MenuScript will attempt to open the font specified in "fontname.font" (the font can be disk-based), and if successfully opened will show you your menu in that font. Placement of menus, items and sub-items is always calculated using the width and height of the current font, so if you don't specify a font, MenuScript defaults to TOPAZ_EIGHTY. Additionally, if this command is used, MenuScript will write out the TextAttr structure of your font and place a pointer to it in the global variable 'MAttr'. The .FONT command may be used only once, but may occur anywhere in the file.

Command .CHECKIMAGE
Format .CHECKIMAGE
Template .CHECKIMAGE
Purpose To define a custom checkmark image.

If you're bored with the old Intuition checkmark imagery, or want a checkmark image that's consistent with the font you're

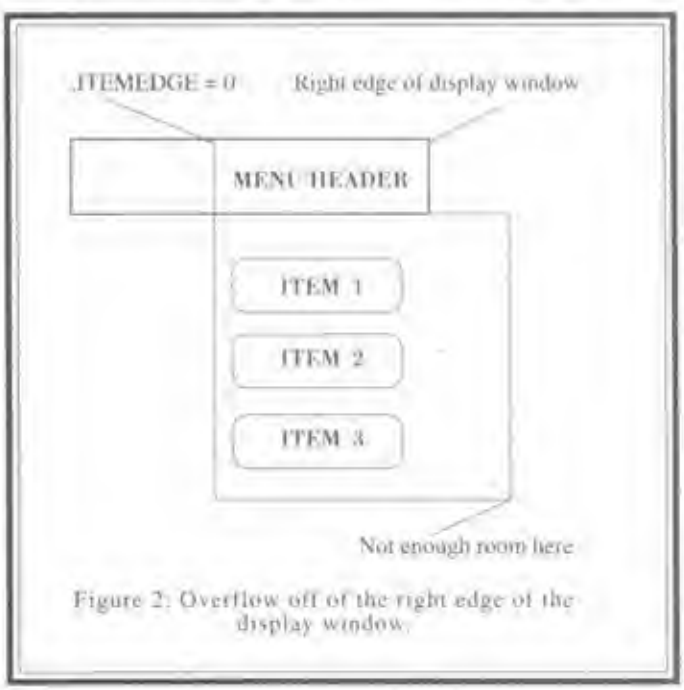


Figure 2: Overflow off of the right edge of the display window.

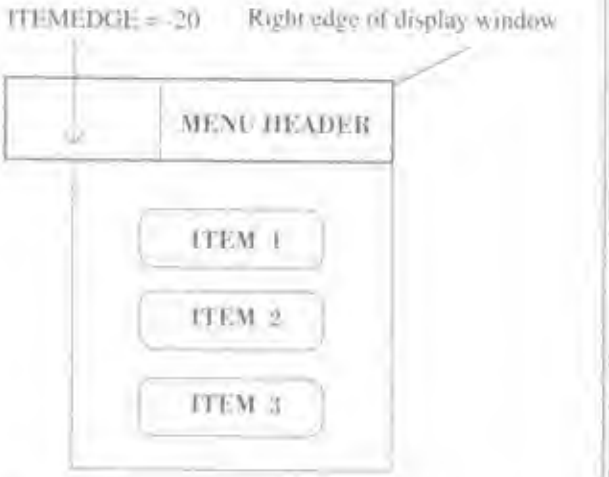


Figure 3: Correction for overflow in Figure 2.

using, then you can use the command as a crude way of constructing a one-bitplane checkmark image. The .CHECKIMAGE command must be on a line by itself; subsequent lines define the actual image. The format of the image data is a series of single lines in quotes consisting of a period (.) where you want a background color pixel and a star (*) were you want a foreground color pixel. Spaces within the quotes are ignored, and the total number of .'s and *'s in each line must be the same multiple of 16 (that is, one of 16,32,58, etc.). The image is ended with a single '#' in quotes on a line by itself. So to create a 16 x 4 checkmark image you could write:

```
.CHECKIMAGE
"********  ********"
;a very boring rectangle
"*.......  .......*"   ;checkmark
"*.......  .......*"   ;the fact you have the
same multiple
"********  ********"   ;as always: 's and *'s
in each line
                       ;otherwise strange
things will happen...
"#"  ;end of image
```

You will, of course, see your checkmark image in your menu, and when the source code is written out it will include the complete image structure. A pointer to that image structure is placed in the global variable 'CheckMarkImage', which you can place directly in the NewWindow->CheckMark field.

If you use Lattice/SAS C or the a 68K assembler, the output source code will contain instructions to correctly place the image data in CHIP RAM. Otherwise you may have to modify the output source code slightly to ensure this takes place. (Image data that is not in CHIP RAM is not displayed, as we all know.)

Command .WIDTH
Format .WIDTH <width>
Template WIDTH <width | *>
Purpose To set the width of the display window.

If your application uses a low-res screen, you may want to design its menu in a lo-res screen. To do this, simply use the .WIDTH command followed by an integer indicating the desired width of the screen and window (.WIDTH applies to both) you wish to use. The default width is 640 pixels (hi-res). .WIDTH followed by a star (*) resets the width to the default value. (Since you use the .WIDTH command only once, you shouldn't need to use .WIDTH *).Widths other than 640 or 320 are not recommended.

Examples:

```
.width 320    ;use a lo-res screen width
.width 640    ;don't need to write this since 640 is the default
.width *      ;restore default width (harmless, but will
              ;cancel any previous .WIDTH commands. Normally
              ;you only use WIDTH once.)
```

Command .HEIGHT
Format .HEIGHT <height>
Template .HEIGHT <height | *>
Purpose To set the height of the display window.

.HEIGHT is similar to .WIDTH and is used in the same way to set the height of the screen and window you wish to use. The default value is 200, and heights other that 200 or 400 are not recommended. .HEIGHT followed by a star (*) resets it to the default.

Command .DEPTH
Format .DEPTH <depth>
Template .DEPTH <depth | *>
Purpose To set the depth of the display window.

Not very useful, but was included for completeness. You can use this command to set the depth of the screen you wish to use. The default value is 2. This command has no effect on the output source code file. .DEPTH followed by a star (*) resets it to the default value.

Command .W_DETAIL
Format .W_DETAIL <detailpen>
Template W_DETAIL <detailpen | *>
Purpose To set the detail pen used by the display window.

If you're experimenting with different windows and window settings, you can use this command to set the NewWindow->DetailPen field of the window in which the menu is displayed. Usually, however, if you play with this value your menus will end up looking worse (if they can be seen at all). The default for this has been carefully set to 2, and you normally won't want to change it. .W_DETAIL followed by a star (*) resets it to the default value.

Command .W_BLOCK
Format .W_BLOCK <blockpen>
Template .W_BLOCK <blockpen | *>
Purpose To set the block pen used by the display window.

.W_BLOCK is similar to .W_DETAIL, and the same warning applies. The default value is 1. .W_BLOCK followed by a star (*) resets it to the the default.

Command .ITEMEDGE
Format .ITEMEDGE <itemedge>
Template .ITEMEDGE <itemedge | *>
Purpose To set the MenuItem->LeftEdge field for each menu item belonging to a particular menu.

It's easier to show how this command works than it is to try and explain it. Figure 1 shows the normal value of the MenuItem->LeftEdge field. Usually it is zero, which means that the leftmost edge of item box is zero, relative to the left edge of the menu title bar. Sometimes, though, there are a lot of menu titles, and by the time you get to the last one, the item box runs off the edge of the window, even though the title bar fits (Figure 2). When this happens (which can be often if you use lo-res screens) the .ITEMEDGE command can be used to push the item box to the left of the menu title box, and back onto the screen (Figure 3). The parameter for .ITEMEDGE is always a negative integer representing the offset in pixels from the menu header. You can have a different .ITEMEDGE for each item box if you wish. .ITEMEDGE followed by a star (*) resets it to the default value, currently 0.

Example:

```
    itemedge -20;start the menu strip 20 pixels to the
                                        ;left of the menu
title box.
```

Command .OFFSET
Format .OFFSET <offset>
Template .OFFSET <offset | *>
Purpose To set the amount of overlap of a sub-item box.

.OFFSET gives you complete control over where a sub-item box overlaps an item box (Figures 4,5,6). The parameter is an integer between -100 and 100. If you specify -100, the sub-item box appears at the leftmost position, relative to the item box. A parameter of 100 will put the sub-item box at the rightmost position relative to the item box. The default value is 75. .OFFSET followed by a star(*) resets it to the default.

Example:

```
    .item ">> Print"
    .offset -50     ;the subi box consisting of "NLQ" and "Draft"
                                ;will overlap about halfway
over the left side
                                ;of the item's select box.
```

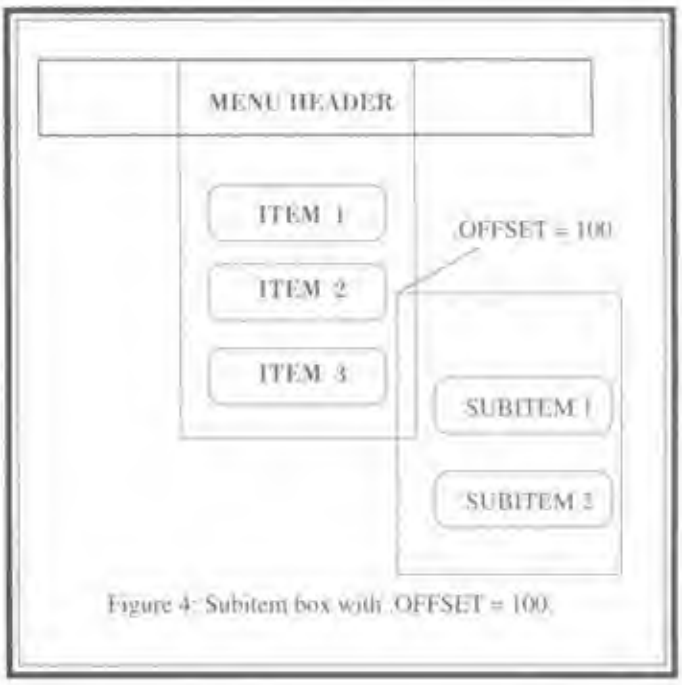


Figure 4: Subitem box with .OFFSET = 100.

```
    .subi "NLQ"
    .subi "Draft"
    .offset *       ;reset for next sub-item box.
```

Command .MODE
Format .MODE <drawmode>
Template .MODE <JAM1 | JAM2 | INVERSVID | COMPLIMENT | *>
Purpose To set the IntuiText->DrawMode field of an item or sub-item.

You can use this command to set the IntuiText->DrawMode field of the IntuiText structures for items and sub-items encountered after this command. It may be used more than once to set different modes for different items. If <drawmode> is INVERSVID, the current mode is OR'd with this value instead of replaced. The default is JAM2. .MODE followed by a star (*) resets it to the default. If you aren't sure what the various drawmodes are, it's best not to use this command.

Example:

```
    .mode JAM1
    .item "Jam 1 item"      ;this item's IntuiText->DrawMode
;field is JAM1
    .mode JAM2
    .item "Jam 2 item"      ;this item's IntuiText->DrawMode
;field is JAM2
    .mode *                                 ;reset to
default
```

Command .IT_TOPEDGE
Format .IT_TOPEDGE <topedge>
Template .IT_TOPEDGE <topedge | *>
Purpose To set the IntuiText->topEdge field for items and sub-items.

This command is used to set the IntuiText->TopEdge field for items and sub-items encounters after the command. The default value is 1, and normally it won't be necessary to change it. .IT_TOPEDGE followed by a star (*) resets it to the default value.

Command .IT_FRONTPEN
Format .IT_FRONTPEN <frontpen>
Template .IT_FRONTPEN <frontpen | *>
Purpose To set the IntuiText->TopEdge field for items and sub-items.

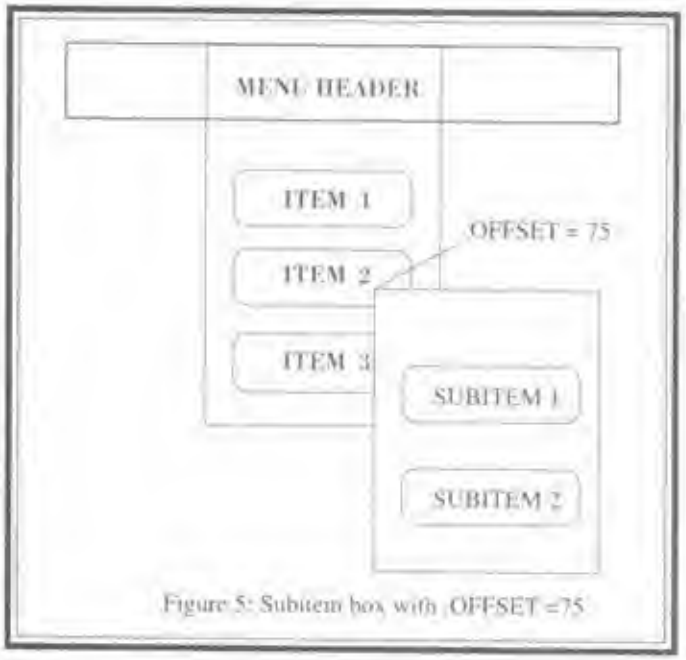


Figure 5: Subitem box with .OFFSET =75

This command sets the IntuiText->FrontPen field for all items and sub-items encountered after it. Its use is identical to .IT_TOPEDGE. The default value is 2, and .IT_FRONTPEN followed by a star (*) resets it the the default.

Command .IT_BACKPEN
Format .IT_BACKPEN <backpen>
Template .IT_BACKPEN <backpen | *>
Purpose To set the IntuiText->BackPen filed for items and sub-items.

You use this command to set the IntuiText->BackPen field for items and sub-items encounters after it. Its use is identical to .IT_FRONTPEN. The default value is 1 and .IT_BACKPEN followed by a star (*) resets it to this.

Command .IT_LEFTEDGE
Format .IT_LEFTEDGE <leftedge>
Template .IT_LEFTEDGE <leftedge | *>
Purpose To set the IntuiText->LeftEdge field for items and sub-items.

This command sets the IntuiText->LeftEdge field for all items and sub-items encountered after you use it. The default value is 2, and * resets it to the default value—useful if you want a menu that allows the user to choose between colors. This is a common menu in many commercial applications and it is easy to construct with MenuScript as follows:

```
; this menu displays four solid color bars for items and
; lets the user select among them

menu "Select Text Color"
    .it_leftedge 20          ;push the left edge of text window to the
                                                    ;right 20 pixels
    .it_backpen 0 ;since we just want blank space of color 0
    ;-------------------------------------------------------------
    ;the following item is just blank space in color 0.
    ;it is checkable and initially checked. When we select
    ;this item (item 0) items 1,2,3 are excluded. We use
    ;box highlighting.
    ;-------------------------------------------------------------
    .item "    " checkable check toggle box exclude 1,2,3 end
    ;
    ;similarly for colors 1,2,3
    ;
    .it_backpen 1
    item "    " checkable toggle box exclude 0,2,3 end
    .it_backpen 2
    item "    " checkable toggle box exclude 0,1,3 end
    .it_backpen 3
    item "    " checkable toggle box exclude 0,1,2 end
    .it_backpen * ;reset to default

;-------------------------------------------------------------
```

Command .A_SOURCE
Format .A_SOURCE
Template .A_SOURCE
Purpose To create assembly language source code for the menu.

This command is identical to the -a switch.

Command .B_SOURCE
Format .B_SOURCE
Template .B_SOURCE
Purpose To create AmigaBasic source code for the menu.

This command is identical to the -b switch.

Command .C_SOURCE
Format .C_SOURCE
Template .C_SOURCE
Purpose To create C source code for the menu (this is the default).

This command is identical to the -c switch.

Parameters

Here are the parameters which may follow an .ITEM or .SUBI command.

Parameter CHECKABLE
Format CHECKABLE
Purpose To make and item checkable.

The item is checkable, but is not initially checked. You are responsible for leaving enough room on the left for a checkmark. Using this parameter is equivalent to setting the CHECKIT flag of the MenuItem->Flags field.—

Command CHECK
Format CHECK
Purpose To initially check an item that is checkable.

If the item is "checkable," then it is initially checked. Using this parameter is equivalent to setting the CHECK flag of the MenuItem->Flags field.

Command TOGGLE
Format TOGGLE
Purpose To be able to toggle the checkmark for this item.

This is equivalent to setting the MENUTOGGLE flag of the MenuItem->Flags field.

Command EXCLUDE
Format EXCLUDE <itemnum>* END
Purpose To select which items are mutually excluded by this one.

For a "checkable" and "toggle" item, this says to mutually exclude each menu item in the <itemnum> list. Items are numbered starting form 0. The list is terminated with the END parameter.

Example:

```
;item "Exclude Item" checkable,toggle,exclude 3,7,1 end
;if we select "Exclude Item", items 3,7, and 1 of this menu
;are unselected
```

Command SELECT
Format SELECT <name>
Template SELECT "Select Text"
Purpose To define alternate text to be displayed when this item is selected.

Instead of complement highlighting, when this item is selected, the text "Select Text" appears.

Example:

```
;when item "Fred" is selected, "...And Barney" appears
item "Fred" select "...And Barney"
```

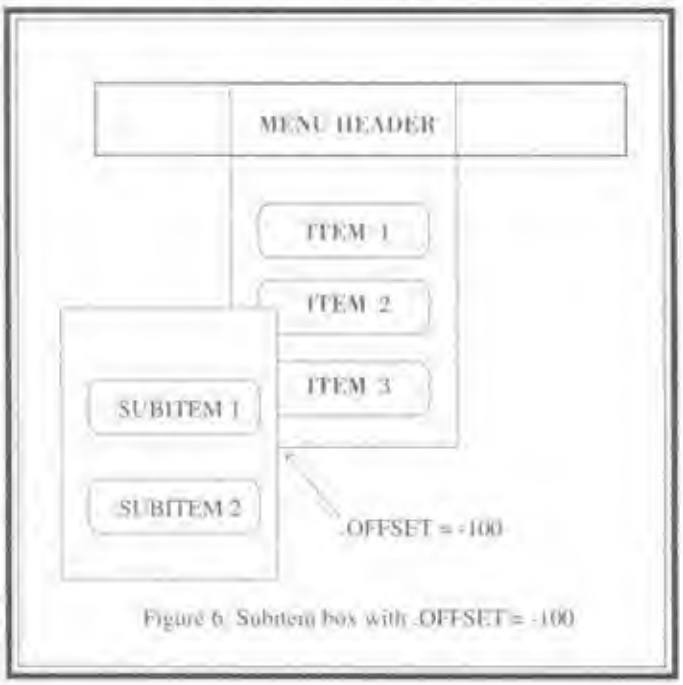


Figure 6: Subitem box with .OFFSET = -100

Command OFF
Format OFF
Purpose To initially ghost this item.

Using this parameter is equivalent to clearing the ITEMENABLED flag of the MenuItem->Flags field.

Command KEY
Format KEY <command key>
Purpose To define a command key equivalent for this item.

MenuScript automatically creates enough room at the end of the item for the command key (hi- or lo-res), so you don't have to worry about leaving enough room on the right. Note that <command key> is not enclosed in quotes.

Example:

```
;this item can be selected by R-Amiga Z
item "Use Z to select this item" key Z
```

Command COMP
Format COMP
Purpose To select complement highlighting for this item.

(This is the default, so you don't have to specify.)

Command BOX
Format BOX
Purpose To select BOX highlighting for this item.

This command is equivalent to setting the HIGHBOX flag of the MenuItem->Flags field.

Command NONE
Format NONE
Purpose To select no highlighting for this item.

Using this command is equivalent to setting the HIGHNONE flag of the MenuItem->Flags field.

Error Messages

When MenuScript detects an error, it stops reading the file and prints the following:

<line number> : <offending line>
ERROR: (error number) <short description of the error>

MenuScript does not specify where in the line the error occurred, but since there is only one command allowed per line, this shouldn't be a problem.

(-1) ER_BREAK Not an error. If you press Control-c at any time (except when the menu is being displayed) MenuScript terminates gracefully with this message.

(0) ER_NO_ERRORS Everything's fine.

(1) ER_NO_INTUITION Major problem. MenuScript was unable to open the intuition.library.

(2) ER_UK_SWITCH MenuScript doesn't recognize your command switch.

(3) ER_NO_INPUT MenuScript was unable to open your input file.

(4) ER_INVALID_TOKEN Your command was not recognized.

(5) ER_NO_ARG There is no numeric parameter for a command which requires one.

(6) ER_NO_SCREEN MenuScript was unable to open a screen to display your menu. This is probably due to low memory.

(7) ER_NO_WINDOW MenuScript was unable to open a window to display your menu. This is probably due to low memory.

(8) ER_2_FONT The .FONT command may be used only once.

(9) ER_NO_GRAPHICS Major problem. MenuScript was unable to open the "graphics.library".

(10) ER_NO_DISKFONT If you use the .FONT command with a disk-based font, MenuScript tries to open the "diskfont.library". You get this error if it was unsuccessful.

(11) ER_NO_FONT The font specified in the .FONT command could not be opened.

(12) ER_NO_MEM MenuScript allocates all memory dynamically, so the number of menus, items, and sub-items is limited only by available memory. When you run out of memory, you get this message.

(13) ER_NO_MODE The parameter given for the .MODE command is not recognized.

(14) ER_ITEM The .ITEM command was used before a .MENU command.

(15) ER_SUBI The .SUBI command was used before an .ITEM command.

(16) ER_NO_QUOTE Missing quote character in a <name> parameter requiring quotes.

(17) ER_INVALID_ARG Parameter is not recognized or is out of range.

(18) ER_INVALID_OFFSET The .OFFSET parameter must be between -100 and 100.

(19) ER_INVALID_ITEMEDGE The .ITEMEDGE parameter must be <= 0.

(20) ER_MISSING_END When you use the .CHECKIMAGE command, your data must be terminated with a "#".

(21) ER_STACK_OVERFLOW This shouldn't happen unless you use more than 64 separate menus at once. But you can have as many items or sub-items for each menu as can be fit on the screen.

Putting It All Together

If you have access to the 1.3 *Amiga ROM KERNAL Reference Manual: Libraries & Devices*, you may want to look at the C source code for the example menu on pages 125-127. Here we will construct precisely the same menu with MenuScript.

For the first menu, the title should be "Project", and the items are "New","Open...","Save","Save As...","Print","About...", and "Quit". Including the command key equivalents, the first lines of our MenuScript program will look like this:

```
.menu " Project "
.item "New"
.item "Open" key O      ;this item has command key equiv 'O'
.item "Save" key S
.item "Save As..."
.item ">> Print"        ;this item has sub-items
```

Now the ">> Print" item has two sub-items, "NLQ" and "Draft" so we continue by writing:

```
.subi " NLQ "
.subi " Draft "
```

And finally:

```
.item "About"
.item "Quit" key Q
```

We're now done with the first menu. The other menus are similar:

```
.menu " Edit "          ;Edit menu
.item "Undo" key Z
.item "Cut" key X
.item "Copy" key C
.item "Paste" key V
.item "Erase All" off   ;this item is ghosted

.menu " Preferences "
.item " Sound "
;"Have your Cake" excludes "Eat it too"
.item "Have Your Cake" checkable check exclude 3 end
;"Eat it Too" exclude "Have Your Cake"
.item "Eat It Too" checkable exclude 2 end
```


If we feed this file to MenuScript, we get the exact menu listed on pages 124-127 of the *RKM*, with considerably less effort on our part. The complete listing for 'rkm.menu' is given in Listing 2.

Inside MenuScript: The Program.

Now that we've learned how to use MenuScript, we look at one of the more interesting aspects of the MenuScript program itself. Essentially what MenuScript does is to translate the instructions found in the input file into a complete Menu structure. Once this is completed, the task of displaying the menu and writing out the source code is relatively simple. The heart of the translation procedure is parsing.

Parsing is the process of splitting complex input into smaller elements which can be processed one at a time. For example, when a C complier encounters an expression like (x+y)/(B-A), it cannot digest the entire expression at once. Rather, it must split this expression into smaller pieces: (,x,+,y,),/,...etc., processing each small piece separately.

Parsing is a vast theoretical field within computer science, and of course there is no single correct way to parse input. Which method is most efficient depends much on the particular application, since what works with one application may not work at all with another. The trick is to select the most efficient method for the type of input that is expected.

So we have to start with a precise definition of exactly what kind of input MenuScript expects. Here there are several choices, and each can lead to a different parsing technique. In MenuScript, we expect a series of statements with (possibly) comments mixed in. A statement can be either a command by itself (.A_SOURCE) or a command followed by one or more parameters (.WIDTH 320). If only one statement is allowed per line, we essentially have a script. The first question we might ask, then, is whether to allow more that one statement per line. Would this make parsing more or less complicated?

(continued on p. 76)

BLIT YOUR LINES

THE FASTEST DRAWS IN THE WEST

by Thomas J. Eshelman

This article consists of an in-depth tutorial concerning the drawing of lines on the Amiga's display screen by means of direct communication with that part of the Agnus chip hardware, commonly referred to as the "blitter." This is not a commonly discussed subject. Therefore, even if you find it a bit outside your present graphics programming requirements, you may still wish to make note of it and file it for future reference.

To Dazzle Is to Boogie

Many programmers writing in different programming languages have effectively used the Move() and Draw() functions from the Amiga graphics library to draw lines on the display. This is fine in the vast majority of situations, but what alternatives exist for someone who wishes to create a more exotic, visual "stunner"? Say, for example, to get the job done, we must draw many thousands of lines each second using some simple algorithm, perhaps employing as input, the output from a random number generator. The results of something like this can be breathtakingly beautiful.

All line draws are handled for you by the blitter. The simple Draw() function contains much underlying code and time-consuming overhead, but eventually, the blitter receives commands it understands and carries them out. If you want to "dazzle the natives," you must avoid this overhead and eliminate all non-essential code. You must learn to cause the blitter to draw lines by directly coercing it to do so. Given the realization that speed means everything, we must consider programming in assembler.

If that last word caused a few heart palpitations, calm down! Assembler is the *simplest* of languages, because it has *no* syntax, as such. You must learn a few dozen, common instructions out of more than 100. You will find many fine assemblers on the market at very reasonable prices, especially when available discounted. In the process, you will develop a much greater understanding of whatever happens to be your high level language of choice. Take your time, and learn new commands as you go. Another good reason to code in assembler is the availability of several powerful, consistent and easily-used symbolic debuggers. My personal preference is the MetaScope by Metadigm. Assembler is made to

TABLE A

BLITTER LINE MODE REGISTERS AND THEIR CONTENTS

Offset from $dff000 Chip Base (in hex)	AMIGA NAMES	Contents or bit settings	Opening remarks
052	BLTAPTL	2 * Sdelta - Ldelta	See Deltas.
048	BLTCPTH	Holds address of word	Rel offset
054	BLTDPTH	where line starts.	screen 0,0
064	BLTAMOD	2 * Sdelta - 2 * Ldelta	See Deltas.
062	BLTBMOD	2 * Sdelta	See Deltas.
060	BLTCMOD	Screen Width	Usually 40
074	BLTADAT	$8000	Constant
072	BLTBDAT	Line mask	Solid = $ff.
044	BLTAFWM	$FFFF	Constant
058	BLTSIZE	Set bits 0-5 at 2. Set bits 6-15 at Ldelta + 1. Ldelta+1 is the line length in points. Ldelta of zero indicates 1024 points. BLTSIZE TRIGGERS DRAW. Load It Last!!	
040	BLTCON0	Bits 12-15 contain the offset of the line's starting point within the 16-bit WORD contained in BLTCPTH and	
	BLTDPTH.	obviously, the latter can be accurate only to within 16 bits. These four bits pin it down.	
		Bits 8-11 are set 1101 respectively. This is a constant value. $b	
		Bits 0-7 are set at $CA respectively. This is the "MinTerm," the value of which causes the line to stay in synch with any mask you put in BLTBDAT. YOU MAY VARY THIS for "effects."	
042	BLTCON1	Bits 12-15 should be set same as 12-15 in BLTCON0 while learining. Bits 7-11 are always set to 0. Bit 6 is set IF 2 * Sdelta < Ldelta. Otherwise it is reset (0). Bit 5 is always set to 0. Bit 4 is the SUD bit. See Octants. Bit 3 is the SUL bit. See Octants. Bit 2 is the AUL bit. See Octants. Bit 1 is set if you want but one point to appear on a display line. (Fills) Bit 0 MUST be set to put the blitter into line draw mode.	

appear difficult by some professional gurus, for the same perverted reason that doctors and lawyers use Latin terms. It makes them stand out as doctors and lawyers—and in our case, gurus.

There is one alternative to writing everything in assembler, but since it may be compiler dependent, I won't address it in depth at this time. It does make sense to write functions in a high level language whenever speed is not essential. These functions can call other functions, which have been written in assembler either because speed is of the essence, or because they will be called a multitude of times during program execution. Arguments can be passed back and forth between the high level functions and the assembler routines on the stack with ease, at least in the case of the C language. For example, in the code accompanying this article, we could have opened the graphics paraphernalia using C functions, thereby saving writing a lot of code. Only the line drawing routine itself must necessarily be in assembler. Note also that it is the only function called repeatedly.

The object of this article is to explain the rudiments concerning how a graphics engine draws lines. This is hardly something we want to relearn every once-in-a-while throughout our lives, so we will create a few, permanent routines that we can use repeatedly. Basically, we want to create real "hotshot" Move() and Draw() routines of our own.

Language of the Blitter

Admittedly, this is not something simple enough to be learned via osmosis. The part that is quite complicated is the "language" of the blitter. This article will not go into detail on how to move data with the blitter, which is its primary *raison d'être*. The writer assumes you know at least a little something about the blitter generally. Rather, we will go into details on using the blitter "Line Mode" to draw—guess what—lines! This Mode is not well-documented in commonly available, Amiga literature. You will find an in-depth discussion of the subject in Dittrich Schemmel's seminal work published by Abacus. Another good source is the file entitled "LineBlit.Arc" written by Mark S. Adams and published by Geodesic Publications. One of the best examples of good assembly programming and blitter techniques may be found in the David Ashley "Tumblin' Tots" tutorial and game code that appeared in *Amazing Computing*,V3.8.

The blitter can draw lines of up to 1024 pixels in length. However, this is not just a matter of describing two points and then connecting them. To master the blitter, we must write a routine that takes the descriptions of two points, translates them as written in "Cartesian talk" (x1,y1 and x2,y2) into "blitter talk," loads the blitter, and causes it to draw. It is essential to know why the routines work so that you may alter them to your own purposes—to draw patterned lines or lines in inverse video.

As you read on, you may realize that the data we input into the blitter really boils down to a starting address in the bitmap indicating where to begin the line, an Octant code indicating in which display area we wish to proceed from that starting point, some Delta data which describes the angle we will take, and the length of the line to draw. "Blitter talk" is just as reasonable as Cartesian values. It is simply "strange" to us amateurs. This seems to cause everyone a temporary, mental block.

Blitter Control Registers

You probably know that the registers of the Amiga's custom chips are "memory mapped." Translated, this means we can use them just as if they actually were addresses in memory. The base of this custom chip "memory" is given what amounts to a fictional address, to wit: 0xdff000 or $dff000. Instead of offsetting from this base by a constant value, say 64, (this is the same as saying $dff040) in order to load or read a particular register, your Amiga compilers and assemblers all use standard #defines or "equates" to translate the cold, meaningless number, (64 or hex 40) into a short, descriptive name, say BLTCON0, for example. Therefore, BLTCON0 would be the same as saying $dff040. You may immediately recognize the string "BLTCON0" as signifying the address of the "first blitter control register." In any event, this is much easier than remembering what data belongs in good, old $dff040! Unfortunately, the Amiga naming scheme for these registers is of far greater significance when the blitter is used to move memory, as it usually does, than it is for when it is used merely to draw lines.

In order to draw lines, we must learn what values are meant to be loaded into these registers set aside by the Amiga engineers as the blitter registers. The difficult part is the esoteric nature of many of these values. At this point, study Table A for a few minutes. It is important that you familiarize yourself with the terms being used in Table A without yet being unduly concerned with what they mean.

Deltas—Coordinate Differences, or Steepness of Slopes

If I am not mistaken, your attention was primarily attracted to all the Deltas of one sort or another encountered when reviewing Table A. The time has arrived to see exactly to what they refer.

The coordinates of a line's starting and ending point are referred to as X1, Y1 and X2, Y2, respectively. These are "Cartesian" values relative to an horizontal "X" and a vertical "Y" axis. In the case of the Amiga, the "X" axis runs from the left edge of the display to the right, and the "Y" axis runs from the top to the bottom edge. Our coordinate values are relative to the top-left point of the display which has the arbitrary value of 0, 0.

For our purposes, DeltaX is said to be an absolute value, equal to the difference of X2 - X1 as defined above. DeltaY is said to be the value equal to the difference of Y2 - Y1. These values must be massaged a bit more before feeding them to the blitter for action.

There are three terms mentioned in Table A to be loaded into the blitter. To compute them, we must first compare DeltaX and DeltaY. The smaller of the two is hereafter referred to as Sdelta (smaller difference value). The larger is hereafter referred to as Ldelta (larger difference value). Now Table A will begin to make sense:

First term = 2 * Sdelta	(Loaded into BLTBMOD)
Second term = 2 * Sdelta - Ldelta	(Loaded into BLTAPTL)
Third term = 2 * Sdelta - 2 * Ldelta	(Loaded into BLTAMOD)

These rather peculiar values are the ones ingested by the blitter in order to properly locate your line. For now, you must either accept them, or acquire a Ph.D. in geometry. I know which

route I took! At least their calculations don't require very much coding.

Before leaving the subject of Deltas, there are two other points worth mentioning. First, if you stop and reflect upon it, Ldelta will be the same as the number of points in the line, less one. The reason it is "less one" is simply because it is computed via subtraction. Thus, by definition, it is one short.

Secondly, and more importantly, in the event that (2 * Sdelta - Ldelta) is negative, the so-called SIGN bit, bit number 6 of BLTCON0 must be set. Take a look right now near the bottom of Table A, so as to cement this into your mind. It is clearly identified by the presence of two asterisks.

Octants—The Companions of the Deltas

The line's "octant" is another unusual component. The blitter is a true graphics engine, and therefore, it needs to know about the octants into which your display area is divided. The blitter sees the overall display divided radially into eight parts. It makes use of a "code," assigned to each of the particular octants, to determine which direction the line is to be drawn.

An octant's corresponding "code" can range from 0 to 7 inclusive, and therefore, can be represented in binary by a combination of three bits. These bits have names. The names are as follows:

TABLE B

RELATIONSHIP OF LINE COORDINATES TO OCTANT CODE

Line coordinates such that:	Octant No.	Code No.	SUD	SUL	AUL
x1 <= x2 y2 <= y1 DeltaY <= DeltaX	0	6	1	1	0
x1 <= x2 y2 <= y1 DeltaX <= DeltaY	1	1	0	0	1
x2 <= x1 y2 <= y1 DeltaX <= DeltaY	2	3	0	1	1
x2 <= x1 y2 <= y1 DeltaY <= DeltaX	3	7	1	1	1
x2 <= x1 y1 <= y2 DeltaY <= DeltaX	4	5	1	0	1
x2 <= x1 y1 <= y2 DeltaX <= DeltaY	5	2	0	1	0
x1 <= x2 y1 <= y2 DeltaX <= DeltaY	6	0	0	0	0
x1 <= x2 y1 <= y2 DeltaY <= DeltaX	7	4	1	0	0

1. Sometimes Up or Down (SUD)
2. Sometimes Up or Left (SUL)
3. Always Up or Left (AUL)

When referring to the binary number formed by these three named bits, SUD is always the most significant bit, and AUL is always the least significant bit.

Observe that these three "friends" get loaded into bits 4, 3 and 2 respectively of the register BLTCON1. Admittedly, here we must do a disproportionate amount of calculation for the number of hardware bits we get to load as the fruits of our effort.

This is a good time to use the data appearing below to draw yourself a diagram to study along with the rest of this subject. Draw a horizontal line, starting at the center of a sheet of paper, extending to the right, and mark it '0 degrees.' Draw from that line, counterclockwise, eight segments of 45 degrees each. While thus drawing this out in circular or radial form on paper, you will familiarize yourself with the idiosyncracies of octants. This may be a nuisance, but bear in mind this is the most eccentric part of drawing lines at warp speed. Furthermore, you will ever have to write them only once. Our object is the creation of a black box routine that you may use forever—or at least more than once.

Segment	Octant No.	Code No.	SUD/SUL/AUL	
0 to 45 degrees	0	6	1 1 0	(bin 6)
45 to 90 degrees	1	1	0 0 1	(bin 1)
90 to 135 degrees	2	3	0 1 1	(bin 3)
135 to 180 degrees	3	7	1 1 1	(bin 7)
180 to 225 degrees	4	5	1 0 1	(bin 5)
225 to 270 degrees	5	2	0 1 0	(bin 2)
270 to 315 degrees	6	0	0 0 0	(bin 0)
315 to 360 degrees	7	4	1 0 0	(bin 4)

Table B shows how we use those Delta calculations described above to relate to the pertinent Octant, and hence values for SUD, SUL and AUL which we will need for blitter loading.

Miscellaneous Bits and Pieces

The line is effectively drawn in 16-bit lengths which repeat throughout the length of the line. You are entitled to prepare a 16-bit mask and "AND" this with each "line piece" as it is drawn, so that the result is a patterned line. This mask is placed inside the BLTBDAT register. $ff or 1111 1111 would produce a solid line whereas 1010 1010 would produce a dotted line, etc.

Bit 1 of BLTCON1 is the "single bit," sometimes known as the "SING" bit. This must be set if you have any intention of utilizing your line as the outline of an area which will be subsequently filled by the blitter. The SING bit being set guarantees that two adjacent pixels shall not be drawn on the same line of the display while drawing in the Line Mode, which otherwise is quite a common occurence. The reason this must be prevented is that, while the blitter appears to draw a "fill" as if it were a "solid area draw," it in fact fills the area line by line. The blitter draws from left to right, across each line of the display, in order to effect the "fill". It senses the area outline and begins the "draw" when encountering the first pixel on a line of the display. It stops "drawing" whenever the next previously drawn pixel on the same display line is encountered. Hence, two successive pixels drawn while in "Line Mode" will cause disaster when encountered by the blitter in a subsequent "fill" mode. It simply won't fill, and it will go totally out of sync.

Bit 0 of BLTCON1 is the LINE bit. This must be set in order that the blitter knows that the data you put in its registers is meant to be used for the purpose of drawing lines, not copying memory blocks. This bit sets the "line draw mode."

The registers BLTCPTH and BLTDPTH are loaded with the address of the word which contains the first point of the line to be drawn. The blitter takes care of shifting this point around within this word, thanks to that mask we extracted from the old coordinate and placed into bits 15-12 of the BLTCON registers. See "Interesting Algorithms," below.

Program Outline

To perform a direct blitter line draw, we first convert a set of coordinates into an Octant code and determine an Ldelta and Sdelta. Secondly, we calculate the address within the bitplane where the line will begin. Then, we must call two routines to prevent other processes from accessing the blitter while we are busy loading it, and finally, we load our computed values along with some system-mandated constant values into appropriate addresses (blitter registers). Beware! The instant the register BLTSIZE is loaded, the line is drawn.

The program begins with a few lines of code that permit us to sense if it was begun from the CLI or the WorkBench. The latter demands a little more overhead that is fully explained in many other sources. With the graphics library opened, we branch to the routine, "playfields." Here, we call the various graphics primitives to produce for us a single playfield upon which we can draw our beautiful lines. Again, this code has been fully delved into by many other writers.

We next enter the loop algorithm out of which we will call "draw_line", our routine of interest. Don't let the algorithm confuse you. What happens is that you get a sequence of varying pattern, full-height lines drawn from one screen diagonal to the other. This is followed by a series of patterned, full-width lines beginning at the diagonal last mentioned and continuing to the diagonal we first started with. Every eighth full sequential series sees a new color register being loaded for a "cycling" effect.

With every full sequence, the hardware is checked to see if the user is depressing the left mouse button. If it is depressed, the code drops into its closing routine where everything allocated is returned and the WorkBench message replied to, in the event this was begun from an icon.

Interesting Algorithms

In order to determine the precise point where the line draw shall begin, we have an inherent problem along the horizontal axis in that the address of the smallest unit we are capable of computing is that of a byte. Luckily, the blitter is capable of shifting around inside a word to the precise bit (pixel) at which it is to draw. If we can compute how many bytes or words from the base address of the bitplane the line is to begin, and how many bits it is to shift once it arrives at the start of that byte or word, the blitter can take it from there, and draw on the precise X1 coordinate requested.

By way of further illustration, consider the fact that all bits in a binary number above the lowest nybble necessarily combine by themselves into a value that is evenly divisible by 8 and 16. Ex: decimal 51 is (0000 0000 0011 0011). The value of the bits above the least significant nybble is 48 decimal. To this, one adds the value in that low nybble, which is decimal 3. We then have the full, decimal equivalent. We want to extract and save the value of this low order nybble. This is done in the program by "AND"-ing it with hex $f. The result is the value we must shift beyond an even byte or word alignment to arrive at the pixel where the line draw is to begin.

Eventually, we will divide the value in X1 by 8 using the asr.w instruction, which will yield a whole, even number of bytes. For example, 51 divided by 8 is 6, meaning our X1 point must be just a tad to the right of byte number 6 in line Y1.

To compute our address, we need only look at Y1 to see how many lines down from the top our line draw begins. Multiply this value by the number of bytes per line. In our lo-res screen this is 40 (40 * 8 is 320). To this value, add your X1 bytes (6 in our example) and, voilà!, you have the number to add to the address of the plane's base to come to the precise address where the line draw begins. The shift value (3) goes into a special register, so the blitter saves you the nuisance of the detail work. The address of the plane's base, of course, is found and stored for us by the early call to the system routine, InitBitMap().

Those of you not familiar with assembly programming should keep your eye open for the progress of this algorithm in the code.

Another thing to observe is that each time the code tests for the deltas by subtractions, a value is moved into register d7. Be apprised that the value moved is more than merely the Octant Code bits. On each occasion, we are setting Bit 0 (the LINE bit) and resetting Bit 1 (the SING bit) in addition to the Octant Code bits, SUL, SUD and AUL. Eventually, you will see that we load the whole thing into BLTCON1 at once, thereby setting many disparate bits.

The final algorithm that bears commenting concerns color indirection. This particular code will clarify any misunderstanding you may have regarding that subject. Observe that a color register value is placed in memory at the location called "fillcolor." We have called for a screen "Depth" of three bitplanes. This allows us eight colors—000 to 111 inclusive. Let us step through the code, assuming we placed a '5' in fillcolor, and that our "colortable" has the value of $00f (dark blue) in the sixth spot of the eight possible.

For the discussion below, let me just mention that the term "MinTerm" refers to a binary value which causes the blitter to impede, permit, or alter data being moved by it through its various DMA channels. You don't need to be overly concerned about this unless you want to move entire blocks of memory using the blitter. Meanwhile, you are free to experiment with the values of MinTerm.

The address of the first bitplane is loaded into d7, and bit 0 in "fillcolor" is tested. It proves to be true ('5' being binary 101). This causes the MinTerm $CA to be loaded into BLTCON0, thus turning on the DMA channels, so that the bits of the first plane will be set wherever the line is to appear. By incrementing d1, we cause the next (second) bit of "fillcolor" to be examined. Bit 1 thereof proves to be false, of course. Therefore, the MinTerm $0A will be loaded into BLTCON0, thus turning off the DMA. The contents of register A2 are incremented so that we now point to the second plane. Now, the bits of this second plane will be zeroed wherever the line is to appear. If you repeat this, you see where the bits in the third plane are once again set. Finally, when the number of the bit being tested (3) is the same as the screen "Depth" contained in d2, the loop quits because the line is totally drawn. To the eye, the combination of the odd planes (1 and 3) being set and the even plane reset will present us with a dark blue line. The value preset in register 101 ($00f) will cause the blue electron gun to turn full on, while shutting off the red and green guns completely. Thus "gunned," our line will be drawn.

Conclusion

Commercial assemblers are relatively inexpensive. They work more quickly than compilers. You can do an edit-compile-run sequence in a few seconds. Once you get in a groove, it will be time to acquire a good symbolic debugger. This tool will enable you to catch programming errors you never knew you were capable of making! I refer to "screwball," unexpected results, not simple system crashes. The former are found only with the greatest of difficulty when programming in a higher level language.

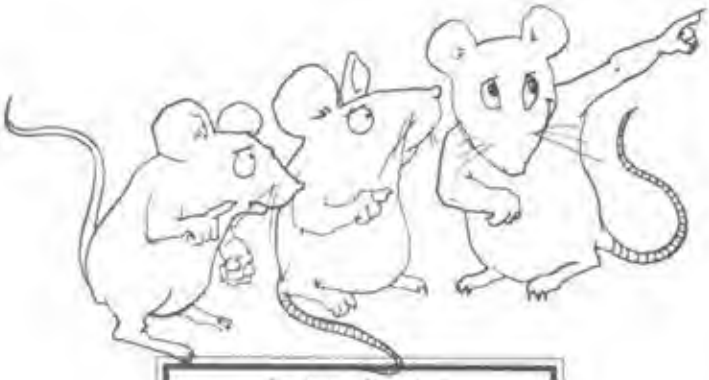
It will be helpful for you to experiment with my code. See for yourself what changes you can bring about. The code is rather sturdy, and will stand a lot of poking about. In any event, it should approximate the very fastest line draws it is possible to achieve. The whole point of this article is for you to excise the "draw_line" routine from the source code for use by yourself in your own programs. I should add that the executable of this program has a size of less than 1500 bytes!

Once you realize that assembler programming is not difficult, and begin to experience the power you have, you will probably find yourself working a lot with Intuition in a high level language but assembling the algorithms that do your "grunt" work. Enjoy!

The Listings for Blit Your Lines can be found on the AC's tech Disk.

Please write to:
Thomas Eshelman
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

A Multitasking Animated Busy Pointer



by Jerry Trantow

If you program on the Amiga, sooner or later you will have an operation that requires the user to wait. If your program is performing a visible or audible action, the user has an indication the program is proceeding and will wait for the action to finish. On the other hand, if your program is accessing a device or performing some lengthy math routines, the user only knows that nothing happens when he continues typing or selecting with the mouse. In these situations, it is helpful to indicate that the computer is busy by changing the pointer to an image that conveys the concept of being busy or waiting. It is relatively easy to change the pointer to the familiar Intuition ZZ cloud or a clock image by calling SetPointer() and ClearPointer(), but it is much more interesting to animate the pointer.

In this article, I will first explain the evolution of routines to animate a busy pointer. Second, I will explain how to replace the SetPointer() function with custom multitasking C routines with very little programming overhead. The third section explains the demo program which allows you to see an animated busy pointer in action and modify the delay and priority. The remainder of the article discusses how the task is initialized and implemented.

Part 1: Evolution of the routines

I always find it interesting to know the background of the problem when exploring a programming example. My experience with implementing busy pointers went through three phases. The code fragments in Examples I, II, and III represent these phases. Example I. Implementing a Busy Pointer.

```
USHORT __chip mypointer[] =
{
    0x0000, 0x0000,    /* reserved, must be NULL */
    0x0400, 0x07C0,    /* 1 word/row image, 2 bit pixels */
    0x0100, 0x0380,
    0x07C0, 0x1FF0,
    0x1FF8, 0x3FFC,
    0x7FFE, 0x0000,    /* end of image */
    0x0000, 0x0000     /* reserved, must be NULL */
};

SetPointer(win, mypointer, 16, 16, -6, 0);
Delay(100);    /* wait 2 seconds */
ClearPointer(win);
```

Example I. Implementing a Busy Pointer

The normal method of implementing a busy pointer is to create a sprite data structure and call SetPointer() when the program enters the busy state and ClearPointer() when the busy state is completed. The sprite structure requires space for some system variables and a block of data defining the colors of each pixel. SetPointer() is an Intuition function that requires the address of the Window the busy pointer is associated with, the address of the sprite data structure, and some sprite dimensions. ClearPointer() is another Intuition function that restores the system pointer to the window.

To go beyond the Intuition ZZ cloud and create the illusion of movement, one needs a method of cycling through a sequence of images while in the busy state. My imagination for this program was stirred by a spinning pointer example I found that required calling a routine every time the image was to change. Creating a sequence of sprite images is an artistic task and limited only by the graphics resolution and your imagination. The array of sprite data that I designed for this program simulates a spinning disk and can be found in file "sprite.img". Example II. Simple Animation of the Busy Pointer.

```
extern SHORT mypointer[];

while (NOT kbhit())
{
    SetPointer(win, mypointer, 16, 16, -6, 0);
    Delay(10);
    ClearPointer(win);
}
```

Example II. Simple Animation of the Busy Pointer

Each time through the busy loop, SetPointer() is called and the sprite index is incremented. As the while loop executes, the result is animation. I used this technique for a short time, but quickly tired of breaking up the busy code to insert SetPointer() calls to obtain a consistent animation action. The rate of action depends on the speed of the program going through the code. In some respect, this dependency is helpful. For instance, if the pointer changes every time a block of data is read from a disk, the pointer will spin faster when reading from a hard drive than from a floppy drive. However, in many cases it is difficult to determine when to change the pointer and discrepancies in animation speed can be distracting. With this in mind, I decided that cycling the pointer images was a job for the Amiga's multitasking abilities.

Example III. Multitasking Animated Busy Pointer

```
[illegible]
```

The original simplicity of starting the busy pointer before the busy code and restoring the pointer when finished is desirable. This is achieved with a task that cycles the pointer with a delay between images and responds to start and stop signals. This approach eliminates inserting SetPointer() calls within the busy code. The priority of the cycling task is used to make the cycle speed either dependent on the execution of the code or a constant speed. If the priority is lower than the busy code or the system is busy, the sluggish pointer animation illustrates this. Whereas, if the priority is set high, the pointer will cycle at a constant rate.

Since I chose my sprite imagery to appear as a spinning ball, the name of the main program file is "SpinTask.c". As stated earlier, the definition of the sequence of sprite images is in file "sprite.img". File "Spawn.c" contains the code for the task that cycles through the pointer images, but more on that later. For now, assume that the task exists and concentrate on how to communicate with it. The listing of main program file SpinTask.c expands on the summary of Example III. Once this program is understood, the routines to implement multitasking busy pointers can be ported to other programs.

PART 2: How to replace SetPointer() with a multitasking animated pointer.

The first step in using these routines is to open a window since pointers are associated with specific windows. When the address of the window is established, a task structure is created and linked to the window by calling:

```
[illegible]
```

Each window that displays the busy pointer needs initialization with this function. Don't worry about multiple windows using the same task server code because each window will have its own task structure. Each time a signal is sent to a task, the task server routine gets called with a stack pointer from the task. In this way, the task server can be servicing several windows and yet, at any given time, it deals only with a single window. The memory overhead for each window is the size of the task structure and task stack.

To simplify the program, I use the window's UserData field to hold the address of the task control block. If your program already uses the window's UserData field, another variable to hold the task address returned by SpawnSpinTask() is required. If SpawnSpinTask() returns NULL, the task creation failed and the task should not be used.

The Spin_usec variable is initialized with the time in microseconds between the pointer images. In NTSC, the standard American display, the screen is updated 60 times per second which corresponds to 16,667 microseconds between updates. Values of 30,000 to 90,000 microseconds are appropriate for Spin_usec. The Priority variable determines the priority of the task and is initialized to 0 by SpawnSpinTask(), which is reasonable for most programming practices. After initialization, the pointer animation is started by sending a signal to the task via:

Signal((struct Task *)WPtr->UserData,SIGF_SPIN), whatever action constitutes the busy action can now begin and the pointer animation will inform and amuse the user while the window is active. When the busy action is complete, a stop signal is sent to restore the system pointer.

```
[illegible]
```

The pointer can be controlled by sending start and stop signals until the program is ready to exit. Before quitting, each task control block should be removed by calling:

```
[illegible]
```

That's all there is to it. In summary—the task is initialized with SpawnSpinTask(). Before starting the busy action, a start signal is sent with Signal(). When the busy action ends, the pointer is restored with a stop signal and the task is removed with KillSpinTask(). To illustrate how easily this can be done, I added seven lines to the Crunchy Frog II article by Jim Fiore to implement an animated busy pointer during his setup_main() function(see file TastyFrog.add)

PART 3: Spintask.c demonstration program.

The rest of SpinTask.c simply initializes the gadgets and waits for gadget messages. The gadgets with ID fields of GAD_DELAYVALUE and GAD_PRIORITYVALUE are integer string gadgets. They can be modified by the user to change the delay between images (how fast the ball spins) and the priority of the task. Selecting GAD_START will check the bounds of the delay and priority variables and signal the task to spin using these parameters. There is no harm sending consecutive start signals. Selecting GAD_STOP will signal the task to stop spinning the pointer. There is also some code to process GADGETUP messages and move between string gadgets.

Since the main program just waits for gadget messages, it doesn't really exercise any busy code. Rather than adding artificial busy code, I left it to the user to run another program while the busy pointer is spinning. Don't forget that the busy pointer is only displayed when the window is active. Try opening a CLI window and typing "dir df0: all" while SpinTask is running with a delay

of 60 milliseconds and priority of -10 or +10. Also, try running your programs and adjust the delay and priority values until all are appropriate. Use these values when you port the busy pointer into your programs. The code is divided into files to enhance portability. The imagery can be replaced with custom images and the routines can be linked to other programs with minimum hassle.

PART 4: How the task works

The basic functions of the task are to call the SetPointer() routine at specific intervals and respond to start, stop, and terminate signals. The task begins by allocating SIGF_START and SIGF_STOP signals. For the terminate signal, I chose to use the SIGBREAKF_CTRL_C signal, which is defined for all tasks.

In order to change the pointer at regular intervals, a timer is needed implement a delay. Since the precision of spinning a pointer is quite low, I decided to open a UNIT_VBLANK timer device using the vertical interrupt. This device has a resolution of +/- 16.67 milliseconds and very low overhead. It doesn't make much sense to change the pointer faster than the screen gets updated.

To use the timer device, you need to set up a timerIOB message linked to a message port. The task sets up these resources before going into a wait state. The remaining trick is to find the window associated with this task control block. Calling FindTask(NULL) returns the address of the task structure and SpawnSpinTask() has initialized the task tc_UserData field to contain the address of the window.

Now it is a simple matter of waiting for either a timer message or a signal. Four signals can occur.

```
(MSG_PORT)      /* call SetPointer() - Call a timer
(SIGF_STOP)     /* cancel timer, ClearPointer()
(SIGBREAKF_CTRL_C) /* terminate the task
(SIGF_PORT)     /* check and handle increment the image and
                   wait a new time)
```

It is not a simple matter because multiple signals can occur while the task is waiting. For example, start and stop signals could occur before the task has a chance to respond or a terminate signal could get sent while the timer is still counting down.

The following sequence works best for me. First, handle the timer message. If it has not expired, cancel it. If it has expired, get the message from the message port. Now, there is a subtle, but important, point. If the signal consists exactly of a timer signal, a new timer should be started. On any combination of signals, a new timer should not be started. At this point, the SIGF_PORT signal is cleared and the remaining signals are handled. The animation is started if exactly the SIGF_START signal is set. If the SIGF_STOP or SIGBREAKF_CTRL_C signals are received in conjunction with SIGF_START, a new animation should not be started. ClearPointer() should be called when either SIGF_STOP or SIGBREAKF_CTRL_C signals are received. Finally, when a SIGBREAKF_CTRL_C signal is received, the function returns all of the resources it allocated.

PART 5: Possible Enhancements

If you can follow this article, you can use the spinning ball as a busy pointer in your programs. You can also design a more clever animation with your own custom pointer imagery. There are other pointer tricks that could make the busy pointer even more interesting. To make the busy pointer even more interesting, cycle the pointer colors or attach another sprite to display more colors. By changing the hot spot, the pounter animation can orbit about the hot spot. The pointer animation could also be used when your code is in a state other than busy.

The technique of using a task to do an auxiliary job is an important part of programming in a multitasking environment. This example shows how to receive signals, use the timer device, and process in the background. The task can be modified to do other jobs. As a final word, don't be afraid to experiment. I recommend keeping the spintask demo program around to quickly determine proper delay and priority values. Have fun!

Listing One

Listing Two

Listing Three

```
/* Listing Three */
```

```
[illegible]
```

Listing Four

```
[illegible]
```

The remainder of the listings for Multitasking Animated Busy Pointer can be found on the AC's TECH Disk

Please write to:
Jerry Trantow
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

—*MenuScript continued from p. 63*

It turns out that if we are using C to parse the input, our task is made slightly easier if we allow only one statement per line. The reasons for this are easy to see. First, there is no need for a special end-of-statement character, for example, the semi-colon of C, so there is one less special character to search for. Second, and more importantly, C already has the special function fgets( ) (found in <stdio>), which reads a single line of a file. Since we know that there is exactly one statement per line, we can use fgets( ) to read each statement from the input file, one after the other.

This point deserves a second look. Suppose we want to allow multiple statements per line, and the semi-colon is treated as an end-of-statement character. Then a typical line of input might look like this:

```
<line 1> menu "Menu 1"; item "item 1"; — comment
<line 2> menu
<line 3>
"Menu 2"; — this is a continuation of <line 2>
<end of file>
```

> MenuScript's design was guided by two philosophies: a language for describing text menus should be simple and easy-to-use and control over fine detail should not be sacrificed for such ease-of-use.

How might we parse this file? Fgets( ) is clearly out, since a single statement could span more than one line. We need to read only one statement at once, and copy that statement into an input buffer. What we could do is write our own statement-grabbing routine, which might look like this:

```
BOOL Grab_a_Statement(FILE input,char *Buffer)
{
    unsigned char input_char;
    do
    {

        input_char = (unsigned char)fgetc(input);
        if (input_char == EOF) return (FALSE);/* end
of file */

        if (input_char == ';') break; /* end of statement
*/

        *Buffer++ = input_char; /* copy into buffer */
    } while (TRUE);
    return (TRUE);
}
```

We could then call Grab_a_Statment( ) until it returned FALSE, and after each call a new statement would be waiting for us in 'Buffer'.

Obviously this is not a great deal of extra code for the convenience of being able to write more than one statement per line. Furthermore, almost all modern computer languages allow multiple-statement lines. MenuScript, though, is highly specialized and not intended to be a general purpose Intuition programming language. It's not the next C++, and so to keep things simple its input files are really script files. This is a good general rule: the more specialized the purpose, the more you might consider making it a scripting language.

Now we come to the most interesting and complex part of parsing. Once MenuScript has read a statement into some kind of input buffer, what next? In MenuScript there can be any amount of white space between the start of the line and the first command, and subsequent parameters are separated by spaces or commas. We might wish to write a function which first skips over leading white space until it comes to a non-whitespace character and saves that position. Our function would then continue to scan over subsequent non-whitespace characters until it found a space or a comma (recall that these characters terminate commands and parameters). It would then return a pointer to the original first non-whitespace character. We might also want it to remember where it is in the input from call to call, and to return NULL when it comes to the end of a line. Such a function would be just what we need to parse each statement, but it would be somewhat tricky to write and debug.

The Good News

Fortunately, we don't have to write this function ourselves. The standard C language implementation contains an obscure function called strtok( ) which can be found buried in <string.h>. This function is precisely what we've been looking for. MenuScript uses strtok( ) extensively when it is parsing input lines.

The first time strtok( ) is called we give it a pointer to a buffer containing the string we wish to parse, and a pointer to a string of terminating characters. Starting at the first character of the buffer, strtok( ) compares each character of the terminating character string to each character in the buffer. When it finds a character that is not included in the terminating character string, it remembers that position in the buffer. It then continues to scan the buffer until a character that is in the terminating character string is found. The terminating character is overwritten with '\0', and a pointer is returned to the original remembered position. If we wish to continue to parse the same line, then we would call strtok( ) again with a 'Buffer' argument of NULL. Strtok( ) returns NULL when there are no more characters in the buffer to look at.

This is actually much simpler than it sounds. We can look at a visual representation of a string buffer and see what successively calling strtok( ) does to it. In the MenuScript language, a line of input might look like this:

<_ width 320 ;comment >

Our string buffer looks like this:

<_ _ w i d t h _ 3 2 0 _ ; c o m m e n t \n>

Here the underscores (_) are spaces, and '\n' is the newline character. We wish to break the input line into the following pieces: <,width> <320> <;comment>. Everything else in the line can be ignored. Let us see how strtok( ) can do this for us.

On a line, a command can be preceded by any amount of white space. Also, a line may be completely blank. So the first call to strtok( ) would be as follows, where Success is a BOOL variable:

Success = strtok(Buffer,"\x20\t\n");

(This example has been simplified for clarity. Although MenuScript uses strtok( ) in much same manner as described here, the surrounding code has been eliminated so as not to obscure the point.) Note that the variable 'Buffer' is a pointer to the string which you want to parse.

Strtok( ) examines the first character of the buffer and compares it to each of '\x20' (space), '\t' (tab), and '\n' (newline). Finding it to be a space ('\x20'), it moves on to the next character, also a space. Finally, after looking at the third character, it finds a '.', which is not one of "\x20\t\n":

<_ _ w i d t h _ 3 2 0 _ ; c o m m e n t \n>

This position is saved. (***EPV—DRAW ARROW***)

Strtok( ) stores a pointer to the '.' character somewhere and continues to scan the string, but this time it will stop when it finds a character that is included in the terminating string. So it skips over the 'w', 'i', 'd', 't', and 'h' characters, stopping when it finds the space immediately after the 'h':

Strtok( ) points 'Buffer' to here (***EPV—DRAW ARROW***)

<_ _ w i d t h _ 3 2 0 _ ; c o m m e n t \n>

Stops here (***EPV—DRAW ARROW***)

Now it overwrites this space with a NULL ('\0') and points 'Buffer' to the period (.) character. The position of this terminating character is also saved so that on the next call strtok( ) can continue to scan from where it left off.

'Buffer' now points here (***EPV—DRAW ARROW***)

<_ _ w i d t h \0 3 2 0 _ ; c o m m e n t \n>

This space is overwritten with '\0' (***EPV—DRAW ARROW***)

Notice at this point *Buffer points to the string: "_width" (no more, no less), which is exactly what we wanted. Strtok( ) has given us the first token of the input string, to process as we please.

In order to grab the next token we call strtok( ) again, but this time with a NULL argument for 'Buffer':

Success = strtok(NULL,"\x20\t\n");

It is the NULL argument which tells strtok( ) that we wish to continue with the original string from the last call.

Strtok( ) starts at the next character after the '\0' and repeats the above procedure. So after a second call to strtok( ), the string buffer looks like this:

'Buffer' now points here (***EPV—DRAW ARROW***)

<_ _ w i d t h \0 3 2 0 \0 ; c o m m e n t \n>

Stops here, and overwrites with '\0' (***EPV—DRAW ARROW***)

And similarly, after a third call:

'Buffer' points here (***EPV—DRAW ARROW***)

<_ _ w i d t h \0 3 2 0 \0 ; c o m m e n t \0>

Stops here (***EPV—DRAW ARROW***)

Since there is no more input, on the fourth call strtok( ) will return NULL, and we are finished parsing this line.

In the MenuScript source code you can find strtok( ) used in all of the sections which read input lines: main( ), GetName( ), ProcessItemArgs( ), and ProcessMenu( ). Note that GetName( ) is used to grab text that is enclosed in quotation marks, so that spaces can be included within quotes.

The listings and all necessary files for MenuScript can be found on the AC's TECH Disk.

Please write to:
David Ossorio
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

AC's TECH Back Issues

AC's TECH Premiere Issue
Volume 1, Number 1

Magic Macros with Resource *by Jeff Lavin*
Reconstructing MFM data from a hard disk, coping with library stubs, creating image data, and a few other tricks that can be done with the Resource disassembler and a few magic macros.

AmigaDOS, Edit and Recursive Programming Techniques *by Mark Purdue*
Creating a hard disk utility using only AmigaDOS commands and the EDIT line editor (on disk).

Building the VidCell 256 Greyscale Digitizer *by Todd Elliott*
Build an 8-bit 256 greyscale digitizer for less than $80. Includes schematics.

An Introduction to InterProcess Communication with ARexx *by Dan Sugalski*
An inside out step-by-step look at what it takes to start working with IPC and ARexx.

An Introduction to the IBm.Library *by Jim Fiore*
Speed development with the dissidents' IFBM FORM-specific library.

Developing a Relational Database in C using dBC III *by Robert Broughton*
Developing a database application using the Lattice dBC III library.

Using Intuition's Proportional Gadgets from FORTRAN 77 *by Joseph R. Pasek*
Using Absoft's FORTRAN 77 to take advantage of most of the Amiga's ROM Kernal without writing extra C or assembly language code.

FastBoot: A Super BootBlock *by Dan Babcock*
FastBoot is a BootBlock that quickly loads an entire disk into memory, creates a RAM: disk, and boots from that RAM: disk.

AmigaDOS for Programmers *by Bruno Costa*
If you want to delete files, find out file sizes, attributes or the amount of disk space, and even run processes from inside your program, read on!

Adapting Mattel's Power Glove to your Amiga *by Paul King and Mike Cabral*
Construct a special cable and write the necessary software in Modula-2 that will interface the Power Glove to the Amiga.

AC's TECH
Volume 1, Number 2

CAD Application and Design: Part I *by Forest Arnold*
A detailed look at the mathematics and programing techniques used in CAD design.

Interfacing Assembly Language Applications to ARexx *by Jeff Glatt*
How to add an ARexx implementation to a program.

Adding Help to Applications Easily *by Philip S. Kusten*
Implement a context-sensitive on line help facility in your applications.

Programming the Amiga's GUI in C—Part I *by Paul Castonguay*
Getting started in C Programming on the Amiga.

Intuition and Graphics in ARexx Scripts *by Jeff Glatt*
Using the ARexx function library rx_intui library which adds a few dozen ARexx commands that allow an ARexx script to utilize Intuition and Graphics routines.

UNIX and the Amiga *by Mike Hubbart*
A different introduction to UNIX for the Amiga programmer.

A Meg and a Half on a Budget *by Bob Blick*
Add 512K of RAM to your 1MB Amiga 500 for about $30.

AC's TECH
Volume 1, Number 3

CAD Application and Design Part II *by Forest Arnold*
Develop an event-driven program which will let us move, resize, and rotate objects.

C Macros for ARexx *by David Blackwell*
Accessing the full power of ARexx from C.

VBROM: Assembly Language Monitor *by Dan Babcock*
Explore your Amiga with this unique and interesting Assembly Language monitor.

The Development of an AmigaDOS 2.0 Command Line Utility *by Bruno Costa*
Using the new features of AmigaDOS 2.0 develop the 'TO' command line utility.

See what you're missing?

Programming the Amiga's GUI in C—Part III *by Paul Castonguay*
Create your first window.

Programming for HAM-E *by Ben Williams*
An introduction to the libraries and techniques required to program for the HAM-E display.

Using RawDoFmt in Assembly *by Jeff Lavin*
Print formatted strings easily and stop wasting time rolling your own code.

Configuration Tips for SAS-C *by Paul Castonguay*
Configure your system for maximum performance with SAS-C.

Accessing the Math Coprocessor from BASIC *by R.P. Haviland*
Using libraries, access the Amiga's math coprocessor from AmigaBASIC.

AC's TECH
Volume 1, Number 4

GPIO—Low Cost Sequence Control *by Ken Hall*
Take control of your Amiga with this Video Toaster-inspired General Purpose Interface.

Programming with the ARexxDB Records Manager *by Benton Jackson*
Learn to use this powerful ARexx database engine by creating a phonebook/auto dialer for use with the popular shareware telecommunications program VLT.

The Development of a Ray Tracer *by Bruno Costa*
The first of a two-part series featuring the theory and application design of a full-featured implementation of an open-ended ray-tracing package.

Programming the Amiga's GUI in C—Part III *by Paul Castonguay*
Paul continues his popular tutorial series with handles, structuring display modules, an introduction to programming graphic images and much more!

Using Interrupts for Animated Pointers *by Jeff Lavin*
Jeff demonstrates a better way to animate pointers as well as the proper use of two types of interrupts.

Language Extensions—Strings of Type StringS *by Jimmy Hammonds*
An introduction to the implementation of strings of type StringS using C constructs.

AC's TECH
Volume 2, Number 1

Spartan—Build your Own SCSI Interface for your Amiga 500/1000 *by Paul Harker*
Stop swapping disks and build this inexpensive and complete hard drive project.

CAD Application and Design Part III *by Forest Arnold*
Develop an architecture for implementing geometric objects as true object-oriented objects.

Implemtnting an ARexx Interface in your C Program *by David Blackwell*
Part one of a structured approach to adding ARexx capabilities to your application written in C.

Programming the Amiga in 680x0 Assembler—Part I *by William P. Nee*
Learn to program the Amiga in 680x0 assembler. Includes A68K assembler on disk!

Programming the Amiga's GUI in C—Part IV *by Paul Castonguay*
The popular programming tutorial continues with faster draw routines.

Programming a Ray Tracer in C—Part II *by Bruno Costa*
The practical use of illumination model theory.

Low-Level Disk Access in Assembly *by Dan Babcock*
Develop an easy-to-use set of routines for performing floppy access without the aid of the operating system.

To order call 1-800-345-3360 now!

Get any *AC's TECH* Back Issue for just $14.95 each!

Also, for a limited time only, The *AC's TECH* Volume 1 (Complete Set) is available for just $45.00!

Fill in the "blanks" in your *AC's TECH* collection!

Developer's Tools

Starting with the next issue, *AC's TECH* for the Commodore Amiga will be featuring a special section for Amiga developers and those who wish to develop commercial products. This section wil give positive insights into the industry as well as much needed information on the latest development tools, steps to a commercial product, licensing, marketing, finances, and business productivity. This section will be devoted to helping you develop your Amiga and your business into a profitable, efficient, and enjoyable investment.

This issue, we are featuring a product section with some of the latest and best programming and development packages available for the Amiga.

AMOS PD
Mark Scott at the Software Exchange is the AMOS PD librarian in America. Hundreds of disks are in the collection, packed full of AMOS Basic code for the beginner or expert. AMOS America is the club name and we work directly with the U.K. librarian. We are also suppliers of AMOS licenseware. Call for more information and a free AMOS PD catalog. 1-9 $4.95 each; 10-24 $3.95; 25+ $2.95. Runs on any Amiga. AMOS Basic programming language.
Software Exchange,1610 George Washington Blvd., Wichita, KS 67211, (316) 685-4763

AMOS-The Creator
A sophisticated, yet easy-to-use development language for the Amiga with more than 500 commands which allow you to access the awesome power of the Amiga. NTSC/PAL compatible. $95
Europress Software, Europa House, Adlington Park, Macclesfield, Cheshire England SK10 4NP, (011) 4462-585-9333

ARexx
Implementation of REXX, a high-level language especially suited for string manipulations and as a macro processor. $49.95
William S. Hawes, P. O. Box 308, Maynard, MA, 01754

AssemPro
Bridge the gap between slow higher-level languages and ultra-fast machine language programming. AssemPro Amiga unlocks the full power of the Amiga's 68000 processor. It's a complete developer's kit for rapidly developing machine language/assembler programs on your Amiga. ISBN 1-55755-030-1 $99.95
Abacus Software, 5370 52nd Street S.E., Grand Rapids, MI 49512, (616) 698-0330

Benchmark Modula-2
Integrated compiler, linker, and EMACS editor. Compiles at 10,000 lines per minute with burst speeds of up to 30,000 lines/minute. Libraries support AmigaDOS, Intuition, Excel, and Modula-2. With 700 pages of documentation. Many demonstration programs. $199.95
Avant-Garde Software, 2213 Woodburn, Plano, TX 75075, (214) 964-0260.

Developer System with Source Debugger and Library Source
The complete Developer System combined with Source Level Debugger and Library Source all for one price. $724.
Manx Software Systems, P. O. Box 55, Shrewsbury, NJ, 07702, (800) 221-0440

Rexx Compiler
Compiles code that is ARexx compatible. It makes the Rexx program faster, and eliminates external library look-up by resolving library function calls and compile time. The Rexx Compiler creates re-entrant code, allowing users to make programs resonant or as part of a support library. Use the Rexx support library, Rexx math library, and Rexx Plus Library functions as built-in functions. Call Rexx programs directly from other languages. Use the compiler listing generated to find constant, symbols, and nesting levels. Find references to symbols, constants, and labels with the cross-reference produced. Find logic errors in the Rexx program by using nesting levels and additional error messages. Find most errors with a single compile. Use Rexx variable interface to access symbols set from host. The programs that have been compiled can be invoked directly from Workbench, from the CLI as a command, or from any host. 2.0 compatible. Available December 1991 $150.
Direct Edwards Group, 19785 West Twelve-Mile Road, Ste. 305 Southfield, MI 48076-2553, (313) 352-4288

SAS/C™ Development System
SAS/C Compiler Development System offers a complete programming environment with SAS/C Compiler, global optimizer, blink overlay linker, LSE screen editor, source-level debugger, comprehensive documentation, and more. Release 5.10 is Version 2 compatible. $300.00
SAS Institute Inc., One SAS Circle Box 8000, Cary, NC 27513-2414, (919) 677-8000

X11 Programmer's System for the Amiga
Develop X11 applications under AmigaDOS. Source code compatible with X11 release 4 or UNIX-based machines. Works with Gfx Base's X11 window system for the Amiga. Contents: X11 R4 include files, xlib athena widjets, xtoolkit, xmu, BSD socket library networks: ethernet DECnet (TSSnet) +CP/IP (NE225), SANA, arcnet and slip. Requires *Lattice 5.10, 3 MB RAM.*
Gfx Base,1881 Ellwell Drive, Milpitas, CA, 95035, (408) 262-1469

AmigaMOP Test Management Package Version 2
The AmigaMOP V2 is a data management and user interface for electronic test systems. Linkable and run-time libraries are included for writing the test programs and data handlers. Other features included in this version are safe memory support, dual multitasking/multiplexing test programs, and a help feature. $349
Go Software, RR#3 Box 442 Spicer Road, Thompson, CT 06277, (203) 923-2348

Source Level Debugger (SDB)
Interactive source-level debugger designed for fast response and ease in debugging. SDB lets user display all function action names; display values of passed parameters; examine values from any active function; customize the debugging environment with reusable command macros and procedures; use function or line-by-line tracing; set breakpoints by lines functions or variables; see actual C source as it executes; and more. $125
Manx Software Systems, P. O. Box 55, Shrewsbury, NJ, 07702, (800) 221-0440

Questions, comments or suggestions? Write to:
Developer's Tools
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

High Resolution Output

from your AMIGA™ DTP & Graphic Documents

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2450 dpi (up to 154 lpi) at a extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

Who are we? We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.

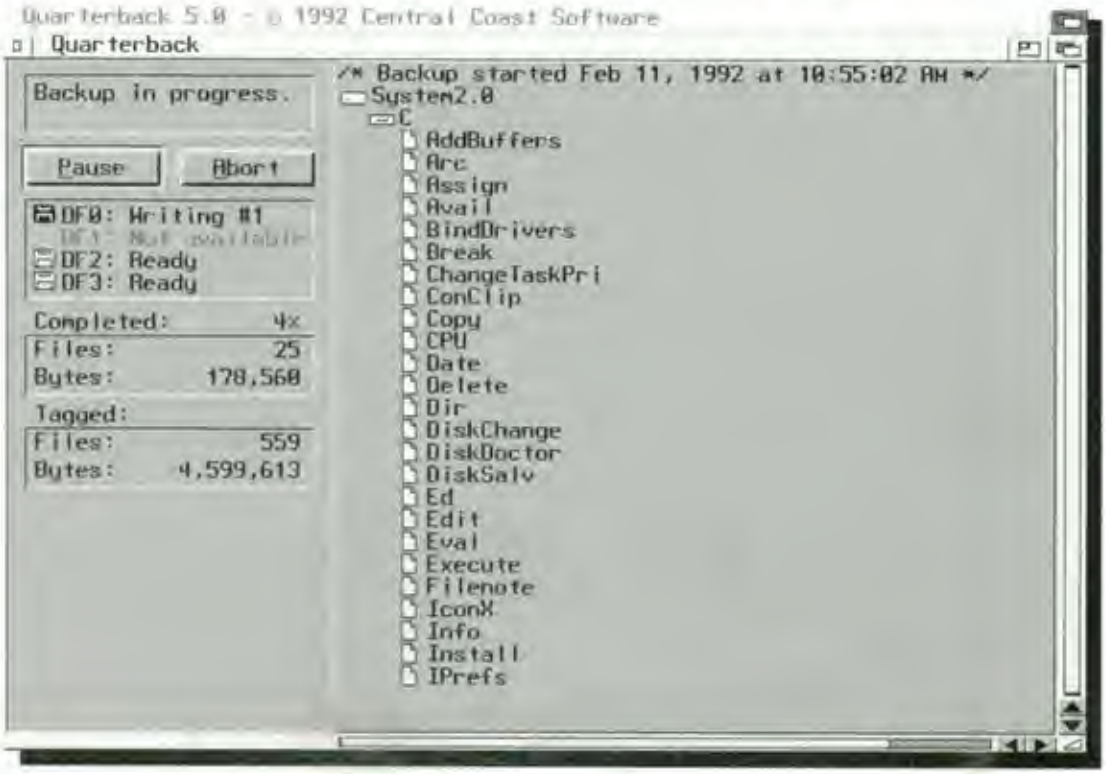
For specific format information, please call.

For more information call 1-800-345-3360

Just ask for the service bureau representative.

Quarterback 5.0

The Next Generation In Backup Software



- *The fastest backup and archiving program on the Amiga!*
- *Supports up to four floppy drives for backup and restore*
- *New integrated streaming tape support*
- *New "compression" option for backups*
- *Optional password protection, with encryption, for data security*
- *Full tape control for retension, erase and rewinding*
- *New "interrogator," retrieves device information from SCSI devices*
- *Capable of complete, subdirectory-only, or selected-files backup and restore*
- *Improved wild card and pattern matching, for fast and easy selective archiving*
- *Restores all date and time stamps, file notes, and protection bits on files and directories*
- *Supports both hard and soft links*
- *Full macro and AREXX support*
- *Full Workbench 2.0 compatibility*
- *Improved user interface, with Workbench 2.0 style "3-D" appearance*
- *Many more features!*

Thousands of people rely on Quarterback for their backup and archival needs. Now, with Quarterback 5.0, there is even more reason to do so. Greater speed, even more features, and proven reliability. And a new "3-D" user interface puts these powerful capabilities at your finger tips. With features like these, it is no wonder that Quarterback is the best selling backup program for the Amiga. Would you trust your data with anything less?

CCS

Central Coast Software

A Division Of New Horizons Software, Inc.

206 Wild Basin Road, Suite 109
Austin, Texas 78746
(512) 328-6650 • FAX (512) 328-1925

Quarterback is a trademark of New Horizons Software, Inc.